Community Experience Distilled

WebGL Game Development

Gain insights into game development by rendering complex 3D objects using WebGL

Sumeet Arora

[PACKT] PUBLISHING

WebGL Game Development

Table of Contents

WebGL Game Development
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Getting Started with WebGL Game Development
    Understanding WebGL
    Differentiating WebGL from the game engine
    Understanding basic 3D mathematics
        Vectors
        Matrices
        Understanding transformations
            Classifying into linear and affine transformations
            Understanding transformations required to render 3D objects
                ModelView transformation
                Projection transformation
    Learning the basics of 3D graphics
        Understanding mesh, polygon, and vertices
        Using indices to save memory
    Understanding WebGL's rendering pipeline
        Framebuffers

- A walkthrough of the WebGL API
  - Initializing the WebGL context
    - Vertex buffer objects – uploading data to GPU
    - Index buffer objects
  - Shaders
    - The vertex shader
      - Vertex transformation with a basic vertex shader
    - The fragment shader
  - Shader variable qualifiers
    - Attributes
    - Uniforms
    - The varying qualifier
    - Compiling and linking shaders
  - Associating buffer objects with shader attributes
  - Drawing our primitive
- Drawing using vertex buffer objects
- Drawing using index buffer objects
- Debugging a WebGL application
- Summary

2. Colors and Shading Languages
- Understanding colors
  - Coloring our square
    - Coloring using the vertex color
  - Learning more about colors
    - Understanding surface normals for lighting calculations
      - Calculating normals from vertices and indices
      - Understanding normal transformation
    - Different types of lights used in games
      - Directional lights
      - Point lights
      - Spotlights
      - Ambient lighting
      - Diffuse reflection (Lambertian reflection)
      - Specular reflection
    - Understanding object materials
- Rendering 3D objects
  - Exporting a 3D object from Blender
  - Understanding and loading the Wavefront (OBJ) format
    - Understanding the material file format (MTL)

Converting the OBJ file to the JSON file format
The JSON file
Parsing the JSON faces array
Loading the JSON model
Rendering without light
Understanding the illumination/reflection model
Lambertian reflectance/diffuse reflection
The Blinn-Phong model
Understanding shading/interpolation models
Flat shading
Gouraud shading
Phong shading
Differentiating the shading models
Implementing Gouraud shading on a Lambertian reflection model
Implementing Gouraud shading – Blinn-Phong reflection
Implementing Phong shading – Blinn-Phong reflection
Summary
3. Loading the Game Scene
Supporting multiple objects
Implementing Face.js
Implementing Geometry.js
Implementing parseJSON.js
Implementing StageObject.js
Implementing Stage.js
Using the architectural updates
Understanding the main code
Understanding WebGL – a state machine
Using mvMatrix states
Understanding request animation frames
Loading the scene
Understanding positional lights
Lighting up the scene with lamps
The vertex shader
The fragment shader
Understanding the main code
Multiple lights and shaders
Adding multiple lamps
The vertex shader
The fragment shader

Implementing Light.js
Applying Lights.js
Understanding the main code
Summary
4. Applying Textures
Texturing basics
Understanding 2D textures and texture mapping
Comprehending texture filtering
Loading textures
A new data type – sampler
Applying a texture to the square
The vertex shader
The fragment shader
Texture wrapping
Testing the texture wrapping mode
The HTML
The event handlers
The redrawWithClampingMode function
Exporting models from Blender
Converting Box.obj to Box.json
Understanding the JSON file with UV coordinates
Parsing UV coordinates from the JSON file
The challenge and the algorithm
Revisiting vertices, normals, and the indices array
Restructuring for texture coordinates
Algorithm one to create new arrays
Algorithm two to create new arrays
Understanding the algorithm using a dry run
Rendering objects exported from Blender
Changes in our JSON parser
Changes in our Geometry object
Loading a textured object
Understanding mipmapping
Implementing mipmapping
Understanding the filtering methods
Nearest-neighbor interpolation
Linear interpolation
Nearest-neighbor with mipmapping
Bilinear filtering with mipmapping

Trilinear filtering
Applying filtering modes
Understanding cubemaps and multi-texturing
Cubemap coordinates
Multi-texturing
Loading cubemaps
Understanding the shader code
Summary
5. Camera and User Interaction
Understanding ModelView transformations
Applying the model transformation
Understanding the view transformation
Understanding the camera matrix
Comprehending the components of a camera matrix
Converting between the camera matrix and view matrix
Using the lookAt function
Understanding the camera rotation
Using quaternions
Understanding perspective transformations
Understanding the viewing frustum
Defining the view frustum
Using the basic camera
Implementing the basic camera
Understanding the free camera
Implementing the free camera
Using our free camera
Implementing the control code
Adding keyboard and mouse interactions
Handling mouse events
Comprehending the orbit camera
Implementing the orbit camera
Understanding the pitch function for the orbit camera
Understanding the yaw function for the orbit camera
Using an orbit camera
Summary
6. Applying Textures and Simple Animations to Our Scene
Applying textures to our scene
Applying a texture to the scene
Implementing the vertex shader code

Implementing the fragment shader code
Working with the control code
Understanding the animation types in 3D games
Understanding time-based animation
Understanding frame-based animation
Implementing time-based animation
Comprehending interpolation
Linear interpolation
Polynomial interpolation
Spline interpolation
A briefing on skinned animation
Using first-person camera
Adding the first-person camera
Improving the first-person camera code
Simple bullet action – linear animation
Reusing objects in multiple bullets
Using B-spline interpolation for grenade action
Using linear interpolation for left-hand rotation
Using texture animation for an explosion effect
Summary
7. Physics and Terrains
Understanding a simple terrain – plane geometry
Rendering our plane geometry
Comparing JavaScript 3D physics engines
Ammo.js
Box2dweb
JigLibJS
Comprehending the physics engine concepts
Updating the simulation loop
Learning about objects in the physics system
Particles
Rigid bodies
Soft bodies
Understanding the physics shapes
Adding gravity and a rigid body to the game scene
Implementing forces, impulse, and collision detection
Diving deep into collision detection
Revisiting the grenade and bullet actions
Cheating in the bullet action

- Extending our terrain with physics
  - Implementing height maps
- Summary

8. Skinning and Animations
- Understanding the basics of a character's skeleton
  - Comprehending the joint hierarchy
  - Understanding forward kinematics
- Understanding the basics of skinning
  - Simple skinning
  - Smooth skinning
    - The binding matrix
    - The final vertex transformation
    - The final normal transformation
- Loading a rigged JSON model
  - Understanding JSON file encoding
  - Loading the rigged model
    - Enhancing the StageObject class
    - Implementing the bone class
    - Implementing the RiggedMesh class
    - Loading the skinned model
- Animating a rigged JSON model
  - JSON model – animation data
  - Loading the animation data
- Exporting models from 3D software in JSON
  - Exporting from Blender
  - Converting FBX/Collada/3DS files to JSON
  - Loading MD5Mesh and MD5Anim files
- Summary

9. Ray Casting and Filters
- Understanding the basic ray casting concepts
- Learning the basics of picking
  - Picking based on an object's color
  - Picking using ray casting
- Implementing picking using ray casting
  - Using a rigid body (collider) for each scene object
  - Calculating the screen coordinates of a click
  - Unproject the vector
  - Creating a ray segment
  - Checking for an intersection

Changing the color of the selected object
Offscreen rendering using framebuffers
Creating a texture object to store color information
Creating a renderbuffer for depth information
Associating a texture and a renderbuffer to framebuffers
Rendering to framebuffers
Applying filters using framebuffers
The vertex shader
The fragment shader
Loading and linking shaders
Understanding the square geometry code
Implementing the filter
Summary
10. 2D Canvas and Multiplayer Games
Understanding canvas 2D basics and the drawing API
Using canvas 2D for textures
Adding 2D textures as model labels
Using the sprite texture
Using a square geometry
Implementing the Sprite class
Implementing the ModelSprite class
Understanding the main flow code
Communicating in real time
Understanding Ajax long polling
Understanding WebSockets
Understanding the WebSocket API
Understanding the WebSockets server
Using Node.js and Socket.IO for multiplayer games
Implementing the HTTP server using Node.js
Understanding Socket.IO
Learning the Socket.IO API
The client socket's reserved events
Understanding Socket.IO rooms
Leaving and joining
Broadcasting an event in the room
Getting the room's information
Storing user data on the server side
Implementing a multiplayer game
Understanding events and the code flow

The code walkthrough
The server code
The client code
Summary
Index

WebGL Game Development

WebGL Game Development

Copyright © 2014 Packt Publishing All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either expressed or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2014

Production Reference: 1180414

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-979-2

www.packtpub.com

Cover Image by Logic Simplified (<sumeet.arora@logicsimplified.com>)

Credits

Author

Sumeet Arora

Reviewers

Jose Dunia

Kevin M. Fitzgerald

Joseph Hocking

Maulik R. Kamdar

Hassadee Pimsuwan

Rodrigo Silveira

Acquisition Editors

Rebecca Pedley

Julian Ursell

Content Development Editors

Chalini Snega Victor

Arun Nadar

Technical Editors

Kunal Anil Gaikwad

Pramod Kumavat

Siddhi Rane

Faisal Siddiqui

Copy Editors

Sayanee Mukherjee

Deepa Nambiar

Project Coordinator

Kranti Berde

Proofreaders

Ting Baker

Simran Bhogal

Maria Gould

Paul Hindle

Indexer

Monica Ajmera Mehta

Graphics

Sheetal Aute

Ronak Dhruv

Disha Haria

Abhinash Sahu

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Sumeet Arora is a tech entrepreneur. He founded Evon Technologies, a consultancy for mobile and web development, and Logic Simplified, a game development company. He holds the position of CTO at Evon and works as a consultant for Logic Simplified. He has worked as an architect consultant for scalable web portals for clients across the globe. His core expertise lies in 3D rendering technologies and collaboration tools. For the past four years, he has been working with various clients/companies on multiplatform content delivery. His own passion towards gaming technologies has helped him help his clients in launching games on various platforms on both web and mobile. Currently his company, Logic Simplified, helps new gaming ideas to launch in the market.

Thanks to my family and colleagues at Evon Technologies and Logic Simplified for assisting me with the graphics and sharing my workload in order to complete the book.

About the Reviewers

Jose Dunia is an experienced web developer with a passion for computer graphics. He would like to see software, especially video games and simulations, being used more within the various levels of education. Jose started developing web projects at the age of 12 and his interest for programming lead him to pursue a B.E. in Computer Engineering at the Universidad Simón Bolívar. He holds an M.S. degree in Digital Arts and Sciences from the University of Florida where he studied Computer Graphics and serious games. Currently, he is working at Shadow Health, a startup company that designs and develops interactive simulations for the advancement of health education.

Kevin M. Fitzgerald is a Platform Architect of Okanjo.com. He has over 12 years of development experience in education, medical systems, and startups and has been tinkering with the web since dial-up modems were mainstream.

Kevin is active in the open source community and has contributed to the Mono project and Umbraco communities. He continues to be active on GitHub, working with the latest technologies and projects.

Kevin and his wife Luciana are celebrating their fifth year of marriage and enjoy long walks on the beach and talking about Node.js, C#, and Bitcoin.

Joseph Hocking is a software engineer living in Chicago, specializing in interactive media development. He builds games and apps for both mobile and web using technologies such as C#/Unity, ActionScript 3/Flash, Lua/Corona, and JavaScript/HTML5. He works at Synapse Games as a developer of web and mobile games, such as the recently released *Tyrant Unleashed*. He also teaches classes in game development at Columbia College, Chicago. His website is www.newarteest.com.

Maulik R. Kamdar is a research scientist working at the intersection of Big Data Visualization, Life Sciences, and Semantic Web. His primary interests revolve around the conceptualization and development of novel, interdisciplinary approaches, which tackle the integrative bioinformatics challenges and guide a bioscientist towards intuitive knowledge exploration and discovery. Maulik has an M.Tech. in Biotechnology, conferred by Indian Institute of Technology (IIT), Kharagpur, one of the most prestigious universities in India. He qualified for the Google Summer of Code scholarship,

an annual program encouraging students across the world to participate in open source projects, for three successive years (2010-12).

He has contributed to Drupal, a content management platform, and the Reactome Consortium, a knowledge base of human biological pathways, on the introduction of HTML5 canvas-based visualization modules in their frameworks. Currently, he is employed at the Insight Center for Data Analytics, Ireland, and researches the application of human-computer interaction principles and visualization methods to increase the adoption and usability of semantic web technologies in the biomedical domain. He has co-authored several scientific publications in internationally acclaimed journals. His recent contribution, titled *Fostering Serendipity through Big Linked Data*, has won the Big Data award at Semantic Web Challenge, held during International Semantic Web Conference, Sydney, in October 2013.

Hassadee Pimsuwan, currently the CEO and co-founder of Treebuild (http://treebuild.com), a customizable 3D printing marketplace and Web3D application. He was working with a Web3D company in Munich, Germany, in 2011 and a web design company in Singapore in 2012-2013. He has graduated in Management Information System from Suranaree University of Technology with first-class honors in 2012.

Rodrigo Silveira is a software engineer at Deseret Digital Media. He divides his time there developing applications in PHP, JavaScript, and Android. Some of his hobbies outside of work include blogging and recording educational videos about software development, learning about new technologies, and finding ways to push the web forward.

He received his Bachelor of Science degree in Computer Science from Brigham Young University, Idaho, as well as an Associate's Degree in Business Management from LDS Business College in Salt Lake City, Utah.

His fascination for game development began in his early teenage years, and his skills grew as he discovered the power of a library subscription, a curious and willing mind, and supportive parents and friends. Today, Rodrigo balances his time between the three great passions of his life—his family, software development, and video games (with the last two usually being mingled together).

I would like to thank my best friend, and the love of my life, Lucimara, for supporting me in my many hobbies, her endless wisdom, and her contagious love for life. I also wish to thank my daughter Samira, who makes each day shine brighter.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

This book is your foray into building in-browser 3D games. The book starts with an introduction to the basics of the low-level 3D rendering API, WebGL. We then transform the low-level API to an implementation of a sample 3D rendering library. We use this library to build components of a concept game, "5000 AD". We walk you step by step from using 3D assets to the implementation of techniques that are used to build a complete in-browser massive multiplayer online role-playing game.

What this book covers

Chapter 1, *Getting Started with WebGL Game Development*, covers basic terminologies of 3D and the basics of WebGL, 3D mathematics, and 3D graphics. It also gives a walkthrough of the WebGL API and discusses the structuring of a WebGL application.

Chapter 2, *Colors and Shading Languages*, explains how to add colors, light, and material to objects in a scene. It discusses how to export 3D objects using tools such as Blender and also explains the basic Wavefront OBJ file format and the JSON file format. Also, we will learn about directional lights in this chapter.

Chapter 3, *Loading the Game Scene*, teaches us how to handle loading and rendering of multiple objects through coding. This chapter also teaches you to add point lights to your scene.

Chapter 4, *Applying Textures*, covers all of the topics on how to create, load, and apply textures to 3D objects. It also teaches advanced techniques such as filtering and cubemaps.

Chapter 5, *Camera and User Interaction*, focuses on evolving our own camera class for our game scene. We will also empower our users to view the game scene from different angles and positions by adding the mouse and keyboard interactivity.

Chapter 6, *Applying Textures and Simple Animations to Our Scene*, starts by simulating a first-person camera and takes it forward by giving weapons to our character. We also dive deep into different animations techniques used in game engines.

Chapter 7, *Physics and Terrains*, explains physics simulation and discusses how physics engines control component trajectories as well as work with collision detection in a game.

Chapter 8, *Skinning and Animations*, covers our study of character animation by understanding the skeleton, which is the base of the character, upon which the body and its motion are built. Then, we learn about skinning and how the bones of the skeleton are attached to the vertices.

Chapter 9, *Ray Casting and Filters*, unveils a very powerful concept in game development called ray casting, which is used to cover cases that cannot be handled by collision detection. This also covers framebuffers, another very important concept used in game development.

Chapter 10, *2D Canvas and Multiplayer Games*, covers the use of 2D rendering in 3D games through canvas 2D, a very powerful 2D drawing API. We also discuss the technology behind HTML-based multiplayer games.

What you need for this book

To learn game development in WebGL, all you need is your favorite text editor and any of the following browsers:

- Mozilla Firefox Version 4.0 and above (http://en.wikipedia.org/wiki/Mozilla_Firefox)
- Google Chrome Version 9 and above (http://en.wikipedia.org/wiki/Google_Chrome)
- Safari Version 6.0 and above (http://en.wikipedia.org/wiki/Safari_%28web_browser%29)

Who this book is for

If you are a programmer who wants to transform the skill of blending imagination and thought in games, this is the book for you. You need to have a good understanding of object-oriented programming, JavaScript, and vector and matrix operations.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Open the `SquareGeomtery.js` file from `client/primitive`."

A block of code is set as follows:

```
<html>
<canvas id="canvasElement"></canvas>
</html>
<script>
this.canvas = document.getElementById("canvasElement");
this.ctx = this.canvas.getContext('2d');
</script>
```

Any command-line input or output is written as follows:

```
#node server.js
#npm install socket.io
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "We can enable the extension in the **Settings** menu in Google Chrome."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/ support and register to have the files emailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Getting Started with WebGL Game Development

We are in 5000 AD and you are wired in to your browser. Rivers have dried up and you are under constant attack by the mutated human race. You are the only hope. You have this book, and your tool is **WebGL**, a JavaScript API for rendering interactive 3D and 2D graphics. Your ancestors, the non-profit technology consortium Khronos Group, gave you this technology. This book has recipes to add rain when you need water, add trees when you need food, generate weapons to fight the mutated Mr. Red, and create your own army of beasts to defend your city. You know the technology that you have is powered by the GPU and is lightning fast. All you need are the recipes.

Wait, hold on a second. Are you a 3D expert? If not, let me teleport you to the current year so you can read this chapter, else stay wired in and move on to Chapter 2, *Colors and Shading Language*.

Since you are reading on, let's focus on the 3D terms that you need to understand for using the recipes in this book. This chapter will cover the following topics:

- Understanding WebGL
- Understanding basic 3D mathematics
- Learning the basics of 3D graphics
- Understanding WebGL's rendering pipeline
- A walkthrough of the WebGL API
- The structuring of a WebGL application and learning shaders for debugging your application

Understanding WebGL

WebGL is a JavaScript API based on OpenGL ES 2.0. OpenGL ES 2.0 is the API for 3D rendering on smartphones running on the iPhone and Android platforms. WebGL enables web content to perform 3D rendering in HTML canvas in browsers that support it. Hence, to learn game development using WebGL, you need to understand JavaScript and HTML basics. If you have an understanding of the mathematics involved in 3D graphics, that's great. However, it is not a must to understand this book. The WebGL program consists of a JavaScript control code and a shader code. The shader code executes on the computer's GPU.

Differentiating WebGL from the game engine

WebGL only provides 3D rendering capability. It is simple, straightforward, and insanely fast. It is good at what it does, that is, rendering 2D and 3D graphics. It is a low-level programming interface with a very small set of commands.

WebGL is not a game engine like Unity, Cocos2D, or Jade. A game engine has many other features, such as collision detection, ray casting, particle effects, and physics simulation. Using WebGL, you can create your own game engine.

WebGL provides functionalities to draw basic primitives such as lines, circles, and triangles that can be used to draw any complex 3D object. It does not provide a direct function to add a camera to your scene. However, a camera class can be evolved to do the same. This is what this book will help you with. It will help you create a library on top of WebGL tailored for creating games and gaming functions.

Understanding basic 3D mathematics

Developing a WebGL game requires a good understanding of 3D mathematics. But we will not cover 3D mathematics in its entirety, since that would require a complete book in itself. In this section, we will cover some basic aspects of 3D mathematics that are required to understand WebGL rendering. We will also build an understanding of the 3D mathematics library that we intend to use in our game. In this section, we will cover a very powerful JavaScript library called **glMatrix** (http://glmatrix.net).

JavaScript or WebGL do not provide any in-built functions for vector and matrix operations. Hence, we use a third-party library to implement them in our code. glMatrix is designed to perform vector and matrix operations in JavaScript and is extremely fast. So, let's walk through the basics of these operations and also understand their corresponding implementation in glMatrix.

Vectors

3D game engines use vectors to represent points in space, such as the locations of objects in a game or the vertices of a polygon mesh. They are also used to represent spatial directions, such as the orientation of the camera or the surface normals of a triangle mesh.

A point in 3D space can be represented by a vector using the *x*, *y*, and *z* axes.

WebGL does not provide any functions for matrix or vector operations. Hence, we use third-party libraries for matrix manipulation.

Let's look at some vector operations provided by one of these libraries:

```
var out=vec3.create()  //Creates an empty vector object.
var out1=vec3.create()  //Creates an empty vector object.
var out2=vec3.create()  //Creates an empty vector object.
var v2=vec3.fromValues(10, 12, 13); //Creates vector initialized
with given values.
var v3=vec3.fromValues(2,3,4);    //Creates vector initialized with
given values.
vec3.cross(out, v2, v3) //Cross product of vector v2 & v3 placed in
vector out.
vec3.normalize(out1, v2) // V2 is normalized. Converted to a unit
vector and placed in out1.
var result=vec3.dot(v2, v3) // Calculates dot product of two
vectors.
var v4= vec4.fromValues(x, y, z, w)// Creates a vector with w value
var v5= vec4(v2, w)
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Matrices

Matrices are primarily used to describe the relationship between two coordinate spaces in 3D mathematics. They do this by defining a computation to transform vectors from one coordinate space to another, for example, object space to world space or world space to view/camera space.

Some useful matrix operations are:

```
var mat=mat3.create() //Creates a new identity mat3
var out=mat3.create() //Creates a new identity mat3
mat3.identity(out) //Set a mat3 to the identity matrix
var result=mat3.determinant(mat) //Calculates the determinant of a
mat3
mat3.invert(out, mat) //Inverts a mat3
mat3.transpose(out, mat) //Transpose the values of a mat3
```

Understanding transformations

You will encounter the word "transformation" in all computer graphics books. This word is mostly used to denote change in the object's state. We can apply scaling, rotation, sheer, or translation transformations to change the state of an object. We can apply a combination of the preceding transformations to change the state of the object. The combinations are generally classified as linear or affine transformations.

Classifying into linear and affine transformations

Linear transformations are the most-used transformations in 3D games. Linear transformations such as scaling, rotation, and sheer will be used throughout your game development career. These transformations are transformations that preserve state, if applied in any order. So, if we scale an object and then rotate it, or first rotate it and then scale it, the end result would be the same. So, transformation is linear, if it preserves the basic operations of addition and multiplication by a scalar.

Some useful functions for linear transformation are:

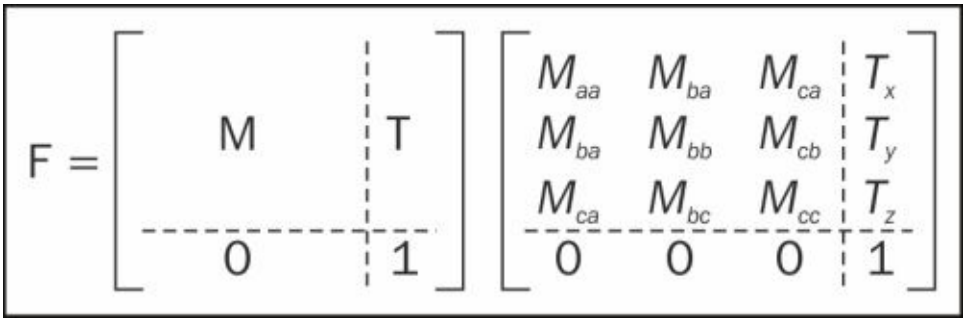
```
var a=mat3.create();
var out=mat3.create();
var rad=1.4; //in Radians
var v=vec2.fromValues(2,2);
mat3.rotate(out, a, rad);  //Rotates "a" mat3 by the given angle
and puts data in out.
mat3.scale(out, a, v)  //Scales the mat3 by the dimensions in the
given vec2
```

An affine transformation is a linear transformation followed by translation. Remember that 3 × 3 matrices are used for linear transformations and they do not contain translation. Due to the nature of matrix multiplication, any transformation that can be represented by a matrix multiplication cannot contain translation. This is a problem because matrix multiplication and inversion are powerful for composing complicated transformations. An example of affine transformation is as follows:

```
var a=mat3.create(); //Identity matrix created
var vertex=vec3.fromValues(1,1,1);
var scale=mat3.create(); //Identity Matrix created
var final=mat3.create(); //Identity Matrix created
var factor=vec2.fromValues(2,2); //Scaling factor of double create
```

```
var factor=vec2.fromValues(2,2); //Scaling factor of double create
2x height and 2x width
mat3.scale(scale,a,factor); // a new scale create after
multiplication
mat3.rotate(final,scale,.4);// new matrix scale created which
contains scaling & rotation
var newVertex=final*vertex;
```

In the preceding code, we created a matrix, `final`, that contained both scaling and rotation operations. We created a composite transformation and applied it on a vertex to get its new position. Now, this `final` `mat3` can be used to transform the vertices of a 3D object. It would be nice if we could find a way to somehow extend the standard 3 × 3 transformation matrix to be able to handle transformations with translation. We can do this by extending our vectors to four-dimensional homogeneous coordinates and using 4 × 4 matrices to transform them. A 4 × 4 matrix is given in the following diagram:



M is the matrix that contains the transformation; the fourth column gives the translation.

Some useful functions for transformations are:

```
var a=mat4.create();//mat4 identity matrix
var final=mat4.create();//mat4 identity matrix
var rad=.5;
var v=vec3.fromValues(0.5,0.5,0.5);
var translate=vec3.fromValues(10,10,10);
mat4.rotateX(final, a, rad) //Rotates a matrix by the given angle
around the X axis
mat4.rotateY(final, final, rad) //Rotates a matrix by the given
angle around the Y axis
mat4.rotateZ(final, final, rad) //Rotates a matrix by the given
angle around the Z axis
mat4.scale(final, final, v) //Scales the mat4 by the dimensions in
```

```
mat4.scale(final, final, v) //Scales the mat4 by the dimensions in
the given vec3
mat4.translate(final, final, v) //Translate a mat4 by the given
vector
```

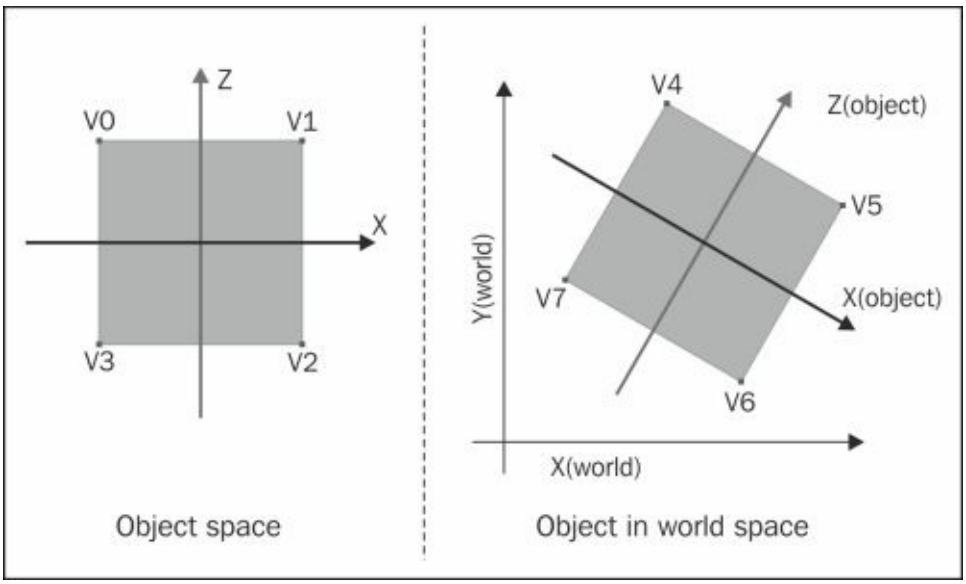
Now, the `final` matrix contains composite transformations of rotation along the axis, scaling, and translation.

Understanding transformations required to render 3D objects

In a game, when we load or initialize a 3D object, the coordinates of different parts of an object are defined with respect to its pivot point. Let us say our designer created a car in Maya and exported the model. When the model is exported, the coordinate of each wheel is defined with respect to the car body. When we translate or rotate our car, the same transformations have to be applied to the wheels. We then have to project the 3D object on a 2D screen. The projection will not only depend on the location of the camera but also on the lens of the camera. In the following section, we will discuss the two types of transformations, ModelView and projection, to help us implement the rendering of the model on the 2D screen.

ModelView transformation

Each model or object we want to draw on the scene has coordinates defined with respect to its own origin and axis; we call it **object space**. When an object is added to the scene and if its own origin coincides with the scene's origin, then its vertices need not be transformed for rendering; but if we want to move the object or rotate it, then we will have to transform its vertices to screen/world coordinates. ModelView transformation is used to transform an object's vertices to world coordinates. An example of this is shown in the following diagram:



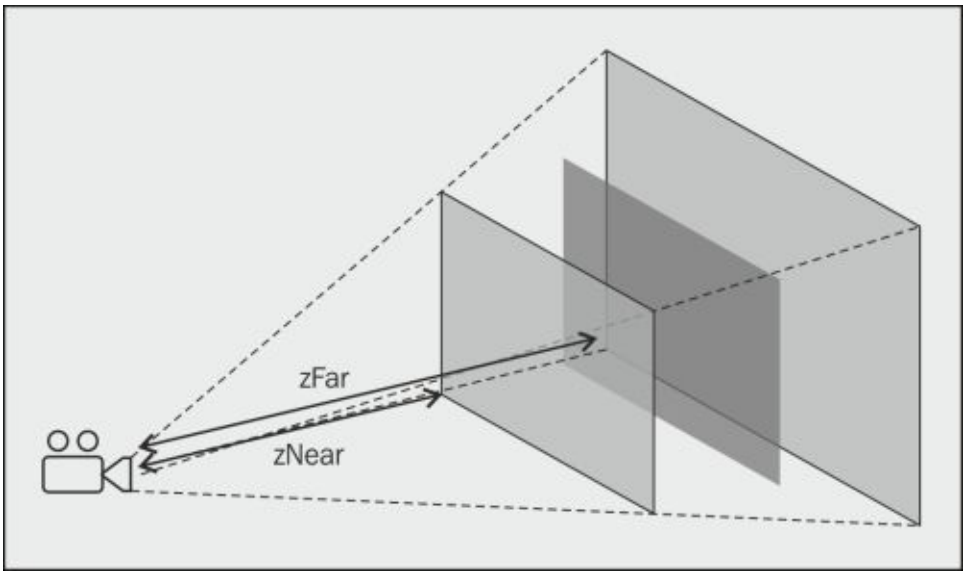
Also, if we move our camera/view around the object, or rotate our camera, then we would need to transform its vertices, changing the world's origin to the camera's position as the origin.

In a nutshell, first model vertices have to be transformed with respect to the scene's origin and then transformed by switching the world's origin to the camera/view position. The final set of all these transformations is maintained as a single 4 x 4 matrix, called the ModelView transformation matrix. The new positions of the model's vertices are obtained via the cross product of each coordinate/vertex of the model with the ModelView transformation matrix, *Vf = Mv * V*. Here, Vf is the final vector, Mv is the [4 x 4] ModelView matrix, and V is the vector corresponding to each vertex. Each coordinate of a vertex is denoted as [*x*, *y*, *z*, *w*], where you can put *w* as 1 for all purposes of this chapter.

Projection transformation

Projection transformation is like setting/choosing the lens of the camera. You want to determine the viewing volume of the scene. You want to determine how objects appear and whether the objects are inside the viewing volume or not. The field of view is the parameter that is used to define the viewing volume. Larger values mean you will cover more objects in your game scene; smaller values are like a telephoto lens (the objects will appear closer than they really are). There are two types of projections; orthographic and perspective. In orthographic

projection, the objects are mapped directly on the screen without affecting their relative sizes. In perspective projection, the distant objects appear smaller. In gaming, we always use perspective projections. The following diagram explains how our primitive is projected on the screen:



Now, to transform vertices with respect to their distance and chosen lens, we use the projection matrix *Vp = P * V*. Here, Vp is the final vector, P is the [4 x 4] projection matrix, and V is the vector corresponding to each vertex.

The following is the code to create a projection transformation:

```
mat4.perspective(out, fovy, aspect, near, far)
```

The parameters used in the preceding code are:

- `fovy`: Field of view
- `aspect`: Scene aspect ratio
- `near`: Near plane to create the clipping region
- `far`: Far plane to create the clipping region

The following code uses the glMatrix library to calculate the perspective matrix using the preceding parameters:

```
var mat=mat4.create()
mat4.perspective(30, gl.viewportWidth / gl.viewportHeight, 0.1,
1000.0, pMatrix);
```

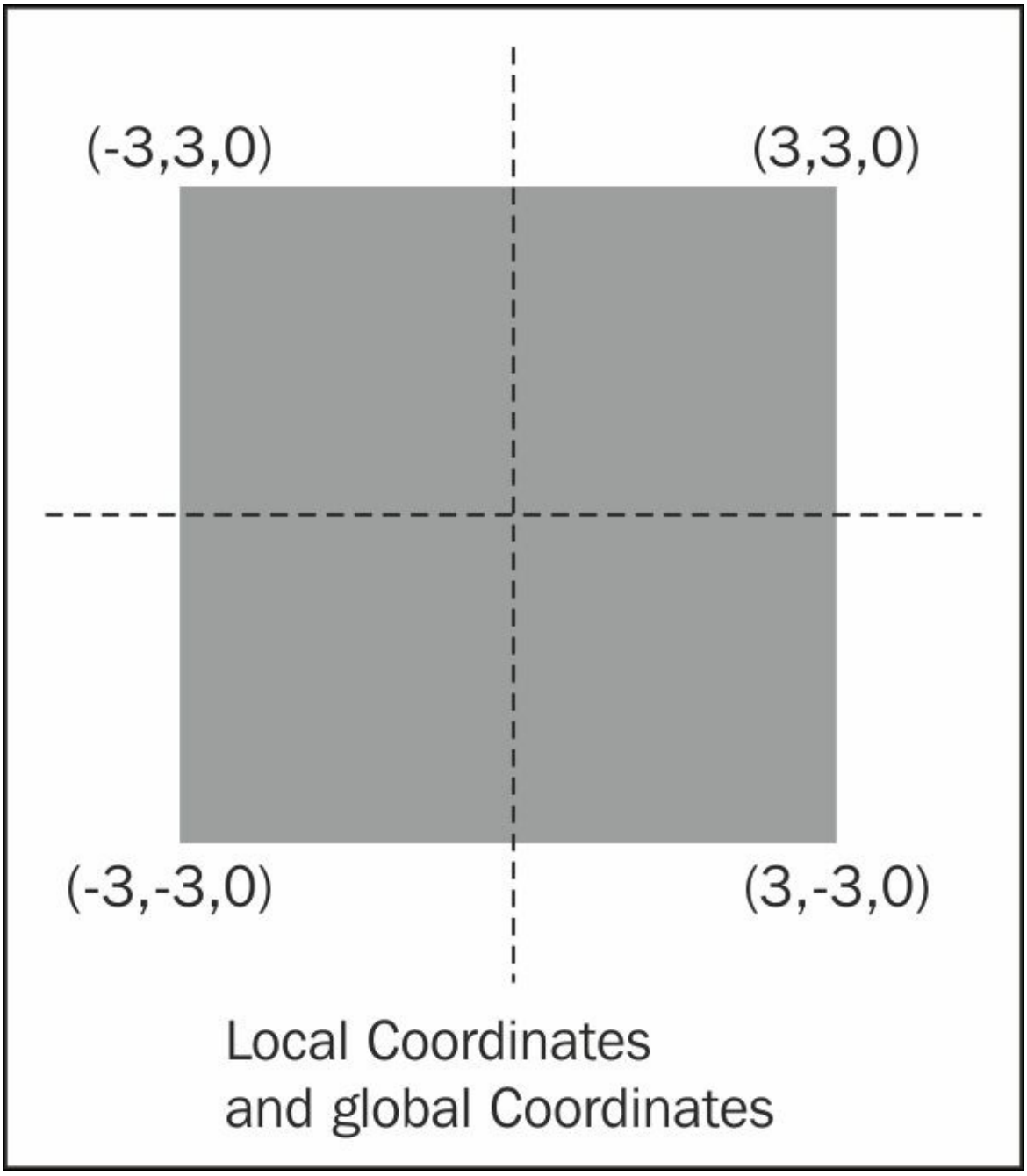
Learning the basics of 3D graphics

We want to save our world in 5000 AD, and we have very little time. So, let's quickly jump into the code and understand the basics along with it.

Understanding mesh, polygon, and vertices

A model in a 3D game is called a **mesh**. Each facet in a mesh is called a **polygon**. A polygon is made up of three or more corners, and each corner is called a **vertex**. The objective of the code provided in this chapter is to render a basic primitive quad. The code creates a mesh with a single polygon and the polygon has four vertices. A polygon with four vertices will form a quad. Each vertex is denoted by a location on the screen. A location on the screen can be represented by using 2 or 3 axes. Each location is defined by using vectors.

In the following example code, we have created an array of vertices with 12 float values (3 per vertex).The following diagram shows the mapping of the coordinates:



The following sample code initializes the vertices:

```
function initBuffers() {
...
vertices = [
             3.0,  3.0,  0.0, //Vertex 0
            -3.0,  3.0,  0.0, //Vertex 1
```

```
             3.0, -3.0,  0.0, //Vertex 2
            -3.0, -3.0,  0.0  //Vertex 3
];
...
}
```

The question might pop up that if we had to draw the mesh of a mutated human, as depicted in the following diagram, then there would have been many polygons and each polygon would have many vertices; so, do we have to define an array with all the vertices by hand?

Well, the mutated human displayed in the preceding screenshot is created using 3D tools such as Maya and Blender. We will export the vertices of the model in a file that is parsed by our JavaScript code, and our program will use the vertices from that file. So, ultimately your code will require vertices, but you will not have to provide them by hand.

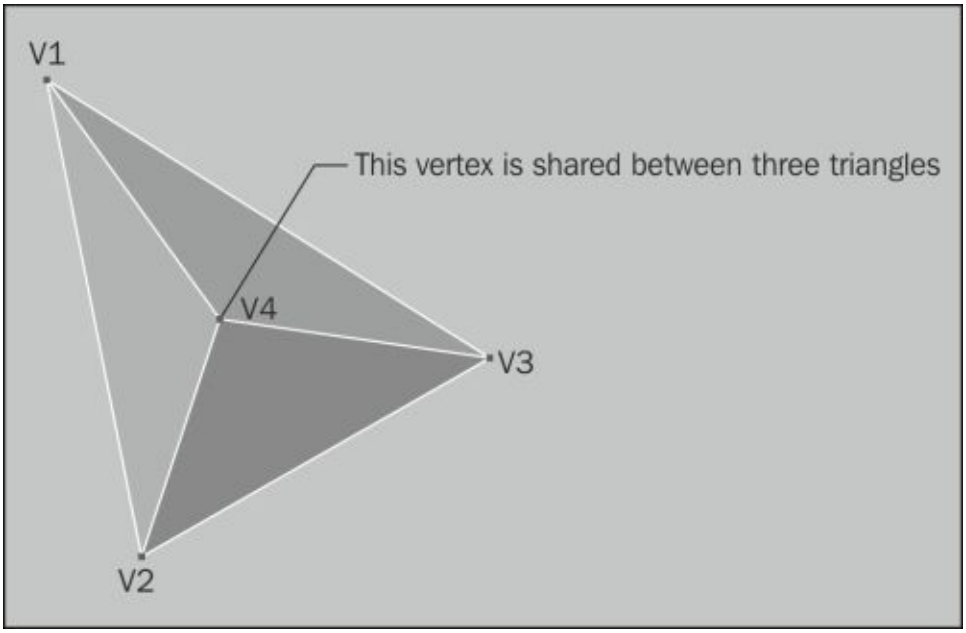
Using indices to save memory

For each vertex that we define, a numerical label is given; for example, "vertex 0" is labeled as "0" and "vertex 1" as "1". These labels are called **indices**. In the following code, we have defined four vertices in the `vertices` array. The next line defines the `indices` array containing numbers [0, 2, 3...], the numbers in the array are labels to each vertex in the `vertices` array:

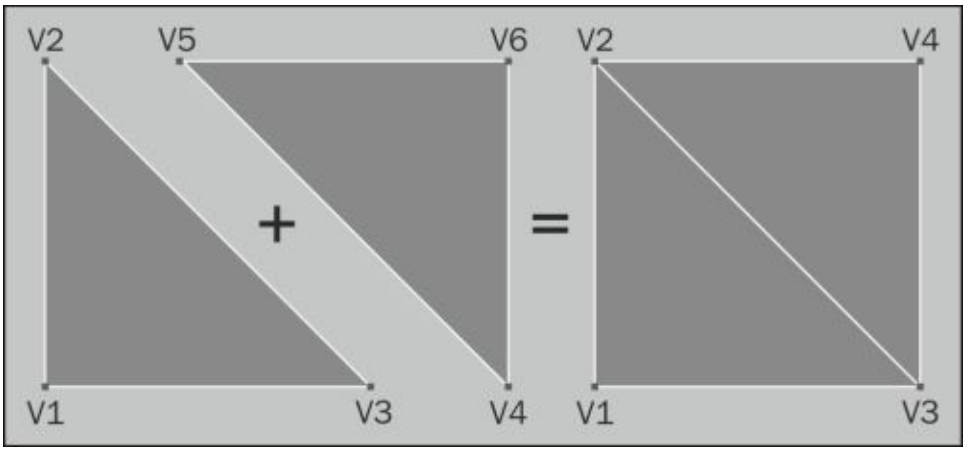
```
function initBuffer() {
vertices = [
             3.0,  3.0,  0.0, //Vertex 0
            -3.0,  3.0,  0.0, //Vertex 1
             3.0, -3.0,  0.0, //Vertex 2
            -3.0, -3.0,  0.0  //Vertex 3
];
indices = [0,2,3,0,3,1];
...
}
```

If we render a sphere, the sphere mesh will be made up of polygons; if we use quads, we will not get a smooth surface. To get a smoother surface, we will require many quads. Hence, we use a polygon with minimum sides. A polygon can have a minimum of three sides, and hence, we prefer triangles. WebGL uses primitives such as points, lines, and triangles to generate complex 3D models.

When we think of a sphere, the vertices will be shared between triangles. We do not want to repeat the shared vertices in our vertex array in order to save memory. The following diagram displays a vertex shared between three triangles:



To explain the concept of shared vertices, we will render a square using two triangles. A triangle will have three vertices; hence, we will need to use 2 * 3 vertices. However, we realize that the two out of three vertices of each triangle are being shared. Hence, we need to declare only four vertices. The following diagram explains how we use two triangles with common vertices to render the **square** geometry:

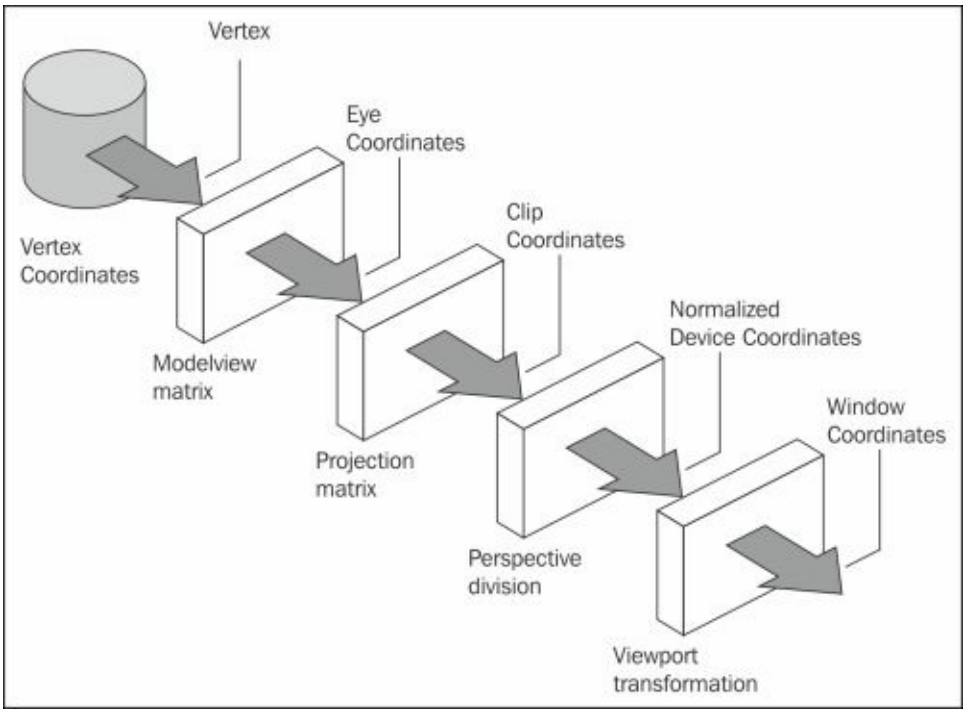


Indices are numeric labels of vertices. They help us inform WebGL on how to connect vertices to form a polygon or a surface. In the preceding example, we

draw one triangle with the vertices [0, 2, 3] and the other triangle with the vertices [0, 3, 1].

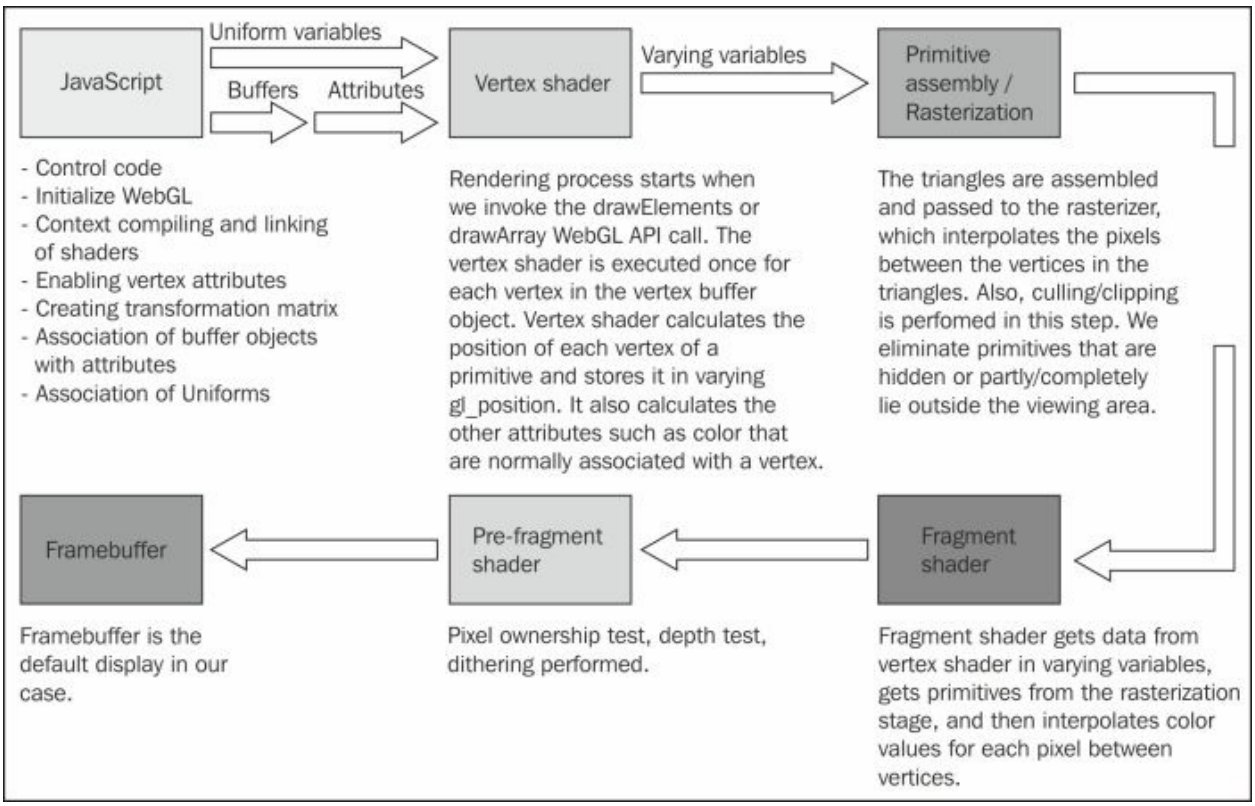
Understanding WebGL's rendering pipeline

The following diagram expresses the actions that WebGL's rendering pipeline needs to perform to project any object on a 2D display screen. The first two steps (ModelView transformation and project transformation) are performed in the vertex shader. Each vertex of an object has to be transformed with respect to its location as well as the viewer's location (camera) on the scene. Then, the vertices that fall outside the viewing area are clipped (perspective divide). Viewport transformation defines the size and location of the final processed object, for example, whether the object should be enlarged or shrunk:



The current GPUs use a programmable rendering pipeline. The earlier graphics card did not allow us to directly change or manipulate the vertices but had built-in functions to rotate or scale them. The programmable rendering pipeline gives us full flexibility to modify vertices of our objects. We can write our own functions to control how our objects are rendered using vertex and fragment shaders. The following diagram describes the different components of the

programmable rendering pipeline. We will cover the details of shaders in the *A walkthrough of the WebGL API* section.


JavaScript
Uniform variables
Buffers
Attributes
Vertex shader
Varying variables
Primitive assembly / Rasterization
- Control code
- Initialize WebGL
- Context compiling and linking of shaders
- Enabling vertex attributes
- Creating transformation matrix
- Association of buffer objects with attributes
- Association of Uniforms
Rendering process starts when we invoke the drawElements or drawArray WebGL API call. The vertex shader is executed once for each vertex in the vertex buffer object. Vertex shader calculates the position of each vertex of a primitive and stores it in varying gl_position. It also calculates the other attributes such as color that are normally associated with a vertex.
The triangles are assembled and passed to the rasterizer, which interpolates the pixels between the vertices in the triangles. Also, culling/clipping is perfomed in this step. We eliminate primitives that are hidden or partly/completely lie outside the viewing area.
Framebuffer
Pre-fragment shader
Fragment shader
Framebuffer is the default display in our case.
Pixel ownership test, depth test, dithering performed.
Fragment shader gets data from vertex shader in varying variables, gets primitives from the rasterization stage, and then interpolates color values for each pixel between vertices.

Framebuffers

A graphics accelerator is the hardware dedicated to drawing graphics; it has a region of memory to maintain the contents of the display screen. Every visible pixel is represented by several bytes of memory called **display memory**. This memory is refreshed a certain number of times per second for a flicker-free display. Graphic accelerators also provide offscreen memory, which is only used to store data. The allocation of display memory and offscreen memory is managed by the **window system**. Also, the part of the memory that can be accessed by WebGL/OpenGL is decided by the window system. There is a small set of function calls that ties WebGL to the particular system. This set of calls supports allocating and deallocating regions of graphics memory, as well as the allocating and deallocating of data structures called graphics contexts, which maintain the WebGL state.

The region of graphics memory that is modified as a result of WebGL rendering is called **framebuffer**. The default framebuffer is provided by the window system and it is the drawing surface that will be displayed on the screen. The calls to create the WebGL drawing surfaces let you specify the width and height of the surface in pixels, whether the surface uses color, depth, and stencil buffers, and the bit depths of these buffers. If the application is only drawing to an onscreen surface, the framebuffer provided by the window system is usually sufficient. However, when we need to render to a texture, we need to create offscreen framebuffers.

The following is the function used to clear/enable color, depth, and stencil buffers:

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

A walkthrough of the WebGL API

This section will explain the basic functions of the WebGL API. We will first understand buffer objects. A WebGL application has two sections: JavaScript control code and shader functions. We will explain the WebGL API functions used in the control code as well as cover the code of a simple shader.

Initializing the WebGL context

To render WebGL 3D content, we need an HTML canvas element. The following HTML code establishes a canvas object that will be used to render 3D content:

```
<canvas id="squareWithDrawArrays" style="border: none;" width="500"
height="500"></canvas>
```

The first thing we do here is obtain a reference to the canvas. We store it in a variable called `canvas`. We pass the `canvas` object to our function `initGL`. This function sets up the WebGL context:

```
var canvas = document.getElementById("squareWithDrawArrays");
initGL(canvas)
```

In the `initGL` function, we obtain a WebGL context for a canvas by requesting the context named `webgl` from the canvas. If this fails, we try the names `experimental-webgl`, `webkit-3d`, and `moz-webgl`. If all the names fail, we display an alert to let the user know that the browser does not have WebGL support. We try different names because different browsers use different names for their WebGL implementation. This is shown in the following code:

```
functioninitGL(canvas) {

  var names = ["webgl", "experimental-webgl", "webkit-3d", "moz-
webgl"];

  for (var i = 0; i<names.length; ++i) {
    try {
      gl = canvas.getContext(names[i]);
    }
    catch (e) { }
    if (gl) {
      break;
    }
  }
  if (gl == null) {
    alert("Could not initialise WebGL");
    return null;
  }
  gl.viewportWidth = canvas.width;
  gl.viewportHeight = canvas.height;
}
```

Vertex buffer objects – uploading data to GPU

A **vertex buffer object** (**VBO**) provides methods for uploading vertex attributes (position, color, depth) directly to the video device for rendering. VBOs offer substantial performance gains because the data resides in the video device memory rather than the system memory, so it can be rendered directly by the video device. Buffer objects can be created using the `createBuffer()` function:

```
vertexBuffer = gl.createBuffer();
```

This only creates the object's name and the reference to the object. To actually create the object itself, you must bind it to the context.

The `bindBuffer()` function is invoked to tell on which of the buffer objects the subsequent functions will operate on. This function is called as follows:

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
```

The target hint can be `ARRAY_BUFFER` (vertex buffers).

There are many allocated buffer objects. We need to specify which buffer object we want to apply the next set of operations on. The `bindBuffer()` function is used to make a particular buffer object the current array buffer object or the current element array buffer object so that the subsequent operations are applied on that buffer object. The first time the buffer object's name is bound by calling the `bindBuffer()` function, the buffer object is allocated with the appropriate default state, and if the allocation is successful, this allocated object is bound as the current array buffer object or the current element array buffer object for the rendering context.

However, the actual memory is only allocated when we invoke the `gl.bufferData()` API call:

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW);
```

The third parameter in the `bufferData` API call specifies how to use the buffer object. The following table explains the different enum values and their usages. In our book, we will use the `gl.STATIC_DRAW` value, but in cases where you might need to animate individual vertices, you will use the `gl.DYNAMIC_DRAW` value:

Enum value	Usage
`STATIC_DRAW`	The buffer object data will be specified by the application once and used many times to draw primitives.
`DYNAMIC_DRAW`	The buffer object data will be specified by the application repeatedly and used many times to draw primitives.
`STREAM_DRAW`	The buffer object data will be specified by the application once and used a few times to draw primitives.

Note

The `gl.bufferData()` API call does not take reference of the buffer object as a parameter. We do not pass the object reference because operations are performed on the current array buffer object or the current element array buffer object.

We can unbind buffer objects using a `bindBuffer()` function call by specifying null as the buffer object parameter:

```
vertices = [
             3.0,  3.0,  0.0, //Vertex 0
            -3.0,  3.0,  0.0, //Vertex 1
             3.0, -3.0,  0.0, //Vertex 2
            -3.0, -3.0,  0.0  //Vertex 3
];
gl.bindBuffer(gl.ARRAY_BUFFER, null);// Deactivate the current
buffer
vertexBuffer = gl.createBuffer();//Create  a reference to the
buffer object
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);//Make the buffer the
current active buffer for memory allocation(Subsequent command)
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW);//Allocate memory for the buffer object.
```

Index buffer objects

Similar to vertex buffer objects, we have index buffer objects to store the indices in the GPU memory. The following code creates a vertex array and an index

array. It then creates the corresponding vertex buffer object and index buffer objects:

```
vertices = [
             3.0,  3.0,  0.0, //Vertex 0
            -3.0,  3.0,  0.0, //Vertex 1
             3.0, -3.0,  0.0, //Vertex 2
            -3.0, -3.0,  0.0  //Vertex 3
];
indices = [0,2,3,0,3,1];
indexBuffer = gl.createBuffer();// Create a reference to the buffer
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);// make the
Index buffer the active buffer notice gl.ELEMENT_ARRAY_BUFFER for
index buffers.
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
gl.STATIC_DRAW);// Alocate memory for the index buffer
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Shaders

A WebGL program is divided in two components; the control code and the shader program. The control code is executed in the system's CPU while the shader code is executed in the system's GPU. Since the control code binds data to the GPU's memory, it is available for processing in the shader.

The vertex shader

The vertex shader is a programmable unit that operates on incoming vertex values and their associated data (normal, color, and so on). The vertex processor usually performs graphic operations such as the following:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

Vertex transformation with a basic vertex shader

The following code is the most basic vertex shader code. In this example, we have provided two inputs to the shader from the control code:

- **Attributes**: Per-vertex data supplied using vertex arrays
- **Uniforms**: Constant data used by the vertex shader

The `aVertexPosition` attribute holds the vertex position and the `mvMatrix` and `pMatrix` uniforms hold the ModelView and the projection matrices respectively, as shown in the following code snippet:

```
<scriptid="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;

  uniform mat4 mvMatrix;
  uniform mat4 pMatrix;

  void main(void) {
    gl_Position = pMatrix  mvMatrix  vec4(aVertexPosition, 1.0);
  }
</script>
```

In the preceding code, we first applied a ModelView transformation on a single vector/vertex. We then applied the projection transformation and set the result to the varying `gl_Position` vector. The varying `gl_Position` variable is declared automatically. The `gl_Position` variable contains the transformed vertex of the object. It is called the **per-vertex** operation, since this code is executed for each vertex in the scene. We will discuss shortly how we pass the vertex position (attribute) and transformation matrix (uniform) to the vertex shader from the control code.

The fragment shader

After the vertex shader has worked on vertex attributes (such as position), the next phase is the **primitive assembly** stage. In the primitive assembly stage, primitive objects such as lines and triangles are clipped and culled. If a primitive object lies partly outside the view frustum (the 3D region that is visible on the screen), then it is clipped; if it lies completely outside, then it is culled. The next phase is **rasterization**. In this phase, all primitives are converted to two-dimensional fragments, also called **pixels**. These fragments are then processed by the fragment shader.

A fragment shader performs the following functions:

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum

The following listed code is the most basic fragment shader. In this code, we simply apply a constant color value to each fragment of our scene:

```
<scriptid="shader-fs" type="x-shader/x-fragment">
  precisionmediump float;

  void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
  }
</script>
```

The `gl_FragColor` variable is declared automatically and the color for each fragment is set. For the purpose of making this shader extremely simple, we did not load any data to the shader; we generally passed these values to the fragment

shader. The values are as follows:

- **Varying variables**: Outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation
- **Uniforms**: Constant data used by the fragment shader
- **Sampler**: A specific type of uniform that represents textures used by the fragment shader

Shader variable qualifiers

Shader variables can be qualified as uniform, attribute, varying, and constant.

Attributes

Attribute-qualified variables are used for data that is passed to shaders frequently. Since every vertex has different attributes, we generally use this qualifier for vertex data. Now in our case (square example), each vertex has a different position, hence, we qualify the `aVertexPosition` vector as the attribute. If each vertex had a different color value, we would have qualified another vector `aVertexColor` as an attribute.

The following line of code qualifies a vector as an attribute in the vertex shader code:

```
attribute vec3 aVertexPosition;
```

The following code gets the reference/index of the shader variable. The variables (attributes) that are declared in the shaders are initialized in the main control code. In order to initialize them, we need to get a reference to them in the control code. The `getAttribLocation` function does just that, as shown in the following snippet:

```
function initShaders() {

  shaderProgram.vertexPositionAttribute =
gl.getAttribLocation(shaderProgram, "aVertexPosition");// getting
the reference to the attribute aVertexPosition from the flow code

}
```

Only floating point scalars, floating point vectors, or matrices can be qualified as an attribute.

Uniforms

The uniform qualifier, just like the attribute qualifier, is used to pass data to the shaders. But the uniform value should not change per fragment/primitive operation. A primitive can have many vertices, hence, a uniform cannot be used to qualify per vertex data. However, a primitive might have the same transformation that can be applied to all vertices, hence, the transformation

matrix is stored in a variable qualified as a uniform.

The uniform qualified variable cannot be modified inside the shader. Also, vertex shaders and fragment shaders have a shared global namespace for uniforms, so, uniforms of the same name will be the same in vertex and fragment shaders. All data types and arrays of all data types are supported for uniform qualified variables.

Let's take a look at how we load uniform data to shaders.

The following lines of code qualify a matrix as a uniform in the fragment shader code:

```
uniform mat4 mvMatrix;
uniform mat4 pMatrix;
```

The following is the code to get the reference of the shader uniform variable:

```
function initShaders() {
……
……
  shaderProgram.pMatrixUniform =
gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform =
gl.getUniformLocation(shaderProgram, "uMVMatrix");
……
……

}
```

The following is the code to load data to a uniform:

```
function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false,
pMatrix);
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false,
mvMatrix);
}
```

Please note that we chose to qualify the transformation matrix as a uniform because we know that the transformation matrix will not change for one particular primitive rendering operation.

The varying qualifier

The varying qualifier is the only way a vertex shader can communicate results to a fragment shader. These variables form the dynamic interface between vertex and fragment shaders. Let's consider a case where each vertex of a primitive, such as a triangle, has a different color and we add light to our scene. Lights might change the color at each vertex. Now, in our vertex shader, we will calculate the new color value of each vertex and share the new color value to the fragment shader using a varying qualified vector; the fragment shader interpolates the color between vertices and sets the color of each fragment/pixel. A vertex shader writes to a varying variable and that value is read in the fragment shader.

Compiling and linking shaders

Shaders exist as independent programs. The following code retrieves the source of these programs by simple JavaScript functions such as `getDocumentElementById`. Then, the three WebGL functions, `createShader`, `shaderSource`, and `compileShader`, initialize and compile the shaders, as shown in the following code snippet:

```
function createShader(gl, id) {
  var shaderScript = document.getElementById(id);
  if (!shaderScript) {
    return null;
  }

  var str = "";
  var k = shaderScript.firstChild;
  while (k) {
    if (k.nodeType == 3) {
      str += k.textContent;
    }
    k = k.nextSibling;
  }

  var shader;
  if (shaderScript.type == "x-shader/x-fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
  } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
  } else {
  return null;
  }

  gl.shaderSource(shader, str);
  gl.compileShader(shader);
```

```
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    return null;
  }

  return shader;
}

function initShaders() {
  var fragmentShader = createShader(gl, "shader-fs");
  var vertexShader = createShader(gl, "shader-vs");
  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);
  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Shaders cannot be initialized");
  }
  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute =
gl.getAttribLocation(shaderProgram, "aVertexPosition");

gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
  shaderProgram.pMatrixUniform =
gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform =
gl.getUniformLocation(shaderProgram, "uMVMatrix");
}
```

The `createShader()` function parses our HTML document tree; it then loads the shader code in the string variable before assigning it to context for compiling. The parent function `initShaders()` first creates the program object and then attaches shaders to the program object. The final step is linking. Our two different shaders need to work together. The link step verifies that they actually match up. If the vertex shader passes data on to the fragment shader, the link step makes sure that the fragment shader actually accepts that input. The two shader sources are compiled into a program, which is passed to the `useProgram()` function to be used. The last few lines of the function get references/indexes to the attribute and uniform variables so that we can later associate attributes to the buffer objects.

Associating buffer objects with shader attributes

We know how to allocate a buffer object and have understood how to get a reference to the shader attribute. Now, we need to associate the buffer object to the vertex shader attribute, so that the following shader code knows where to load its data from. The first line of the following code makes the buffer that we need to associate as the current buffer. The second line associates the current buffer (`squareVertexPositionBuffer`) with the shader attribute (`shaderProgram.vertexPositionAttribute`):

```
gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
squareVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

The parameters of the function `glVertexAttribPointer` are defined as follows:

```
void glVertexAttribPointer(Index,Size,Type,Norm,Stride,Offset)
```

The parameters of the function `glVertexAttribPointer()` are explained as follows:

- `Index`: This parameter specifies the reference to the vertex.
- `Size`: This parameter specifies the number of components specified in the vertex. Valid values are 1 to 4.
- `Type`: This parameter specifies the data format. Valid values are `BYTE`, `UNSIGNED_BYTE`, `FLOAT`, `SHORT`, `UNSIGNED_SHORT`, and `FIXED`.
- `Norm`: This parameter is used to indicate whether the non-floating data format type should be normalized when converted to a floating point value. Use the value `false`.
- `Stride`: The components of the vertex attribute specified by size are stored sequentially for each vertex. The `Stride` parameter specifies the delta between data for vertex index I and vertex (I + 1). If `Stride` is 0, the attribute data for all vertices is stored sequentially. If `Stride` is >0, then we use the stride value as the pitch to get vertex data for the next index.
- `Offset`: This is the byte position of the first element of each attribute array in the buffer. This parameter is used to create interleaved arrays.

Drawing our primitive

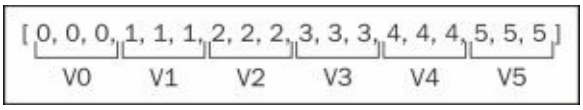
We need to understand two drawing functions that we will use frequently. One of them is the `drawArray()` function:

```
gl.drawArrays(Mode, Offset, Count)
```

The parameters of the `drawArray()` function are explained as follows:

- `Mode`: This relates to the primitives you would like to render. Valid values are `POINTS`, `LINES`, `LINE_STRIP`, `LINE_LOOP`, `TRIANGLES`, `TRIANGLE_STRIP`, and `TRIANGLE_FAN`.
- `Offset`: This is the starting vertex index in the enabled vertex array.
- `Count`: This is the number of indices to be drawn.

The `Mode` parameter needs further explanation. When drawing 3D primitives/models, we pass the vertices array. While drawing, the interconnection of vertices is decided by the `Mode` parameter. The following diagram shows the array of vertices in the actively bound buffer:

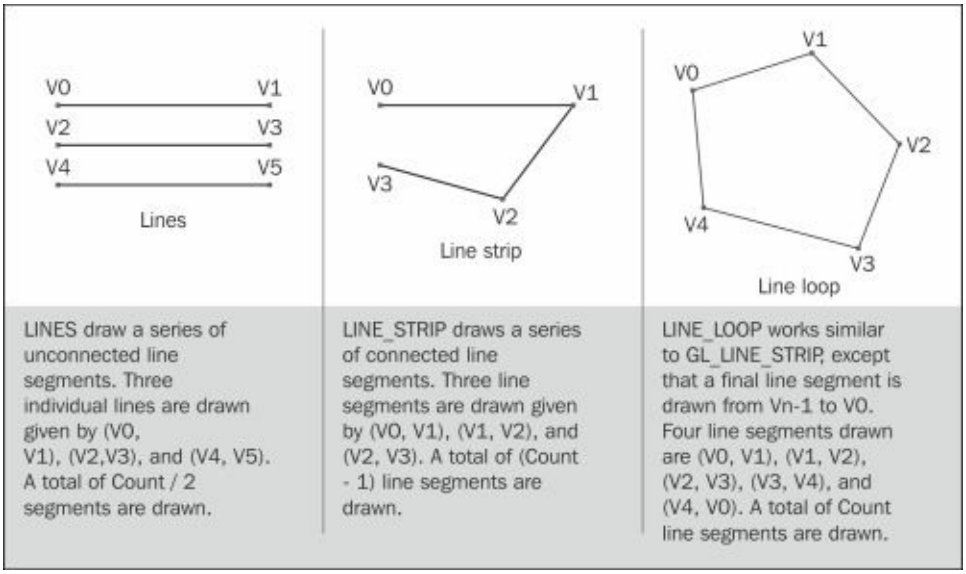


The `drawArrays()` function will treat the vertices of the preceding diagram based on the `Mode` parameter. If you want to draw a wire mesh, you will use the `LINE_XXXX` mode values. If you want to draw solid geometries, you will use the `TRIANGLE_XXXX` mode values.

We have listed the `drawArrays` function call with different mode values, as follows:

```
gl.drawArrays(gl.LINES, 0, vertexPositionBuffer.numItems);
gl.drawArrays(gl.LINE_STRIP, 0, vertexPositionBuffer.numItems);
gl.drawArrays(gl.LINE_LOOP, 0, vertexPositionBuffer.numItems);
```

The following diagram shows how WebGL will draw the same set of vertices with the `LINE_XXX` option:



LINES draw a series of unconnected line segments. Three individual lines are drawn given by (V0, V1), (V2,V3), and (V4, V5). A total of Count / 2 segments are drawn.

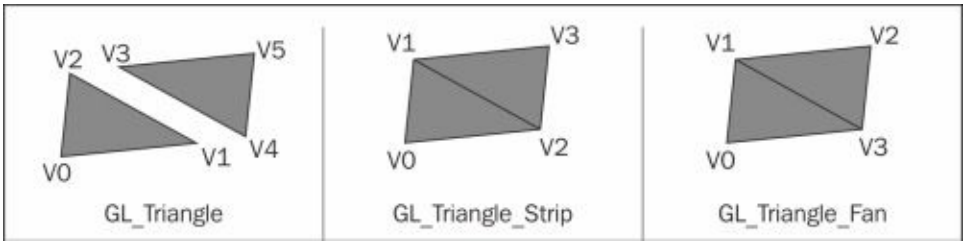
LINE_STRIP draws a series of connected line segments. Three line segments are drawn given by (V0, V1), (V1, V2), and (V2, V3). A total of (Count - 1) line segments are drawn.

LINE_LOOP works similar to GL_LINE_STRIP, except that a final line segment is drawn from Vn-1 to V0. Four line segments drawn are (V0, V1), (V1, V2), (V2, V3), (V3, V4), and (V4, V0). A total of Count line segments are drawn.

The following code shows the different parameters passed to the `drawArrays` function to draw geometries:

```
gl.drawArrays(gl.TRIANGLES, 0, vertexPositionBuffer.numItems);
gl.drawArrays(gl.TRIANGLE_STRIP, 0, vertexPositionBuffer.numItems);
gl.drawArrays(gl.TRIANGLE_LOOP, 0, vertexPositionBuffer.numItems);
```

The following diagram shows how WebGL will draw the same set of vertices with the `TRIANGLE_XXX` option:



TRIANGLES draw a series of separate triangles. Two triangles given by vertices (V0, VI, V2) and (V3, V4, V5) are drawn. A total of Count / 3 triangles are drawn.

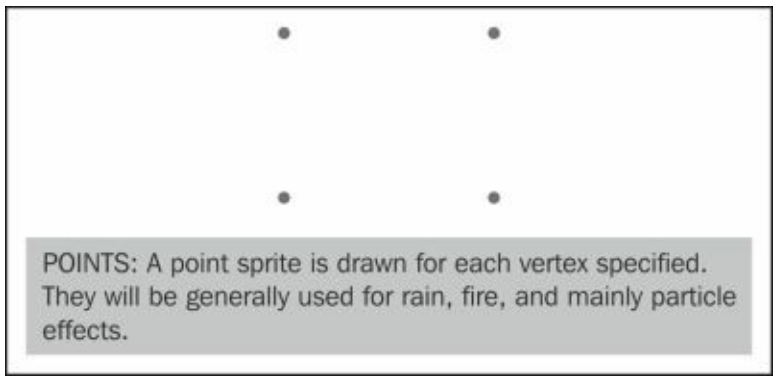
TRIANGLE_STRIP draws a series of connected triangles. Three triangles are drawn given by (V0, V1, V2), and (V2, V1, V3) (note the order). (Count - 2) triangles are drawn.

TRIANGLE_FAN also draws a series of connected triangles. The triangles drawn are (V0, V1, V3), and (V1, V2, V3). (Count - 2) triangles are drawn.

The following function shows the usage of `drawArrays` with the `gl.POINTS` mode value:

```
gl.drawArrays(gl.POINTS, 0, vertexPositionBuffer.numItems);
```

The following diagram shows how WebGL will draw the same set of vertices with the `POINTS` option:


POINTS: A point sprite is drawn for each vertex specified. They will be generally used for rain, fire, and mainly particle effects.

The `drawArrays` function is effective when we have simple geometry; for complex geometry, we use the `drawElements` function call:

```
gl.drawElements(Mode, Count, Type, Offset)
```

The `drawElements` function uses the index buffer object and the `drawArrays` function uses the vertex buffer object to read the data. The parameters of the `drawElements` function are explained as follows:

- `Mode`: This is similar to the `drawArrays` function.
- `Count`: This specifies the number of indices to draw.
- `Type`: This specifies the type of element indices stored in indices. Valid values are `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, and `UNSIGNED_INT`.
- `Offset`: This is the starting vertex index in the enabled vertex array.

Drawing using vertex buffer objects

Please open the file `01-SquareWithDrawArrays.html` in your favorite text editor. The HTML page has three script sections:

- The script section marked with `id="shader-fs"` and `type="x-shader/x-fragment"` contains the fragment shader code
- The script section marked with `id="shader-vs"` and `type="x-shader/x-vertex"` contains the vertex shader code
- The script section marked `type="text/JavaScript"` contains the control code

The HTML code invokes the `start()` function on loading its body. The `start()` function performs the following functions :

- Gets the canvas object using the `document.getElementById("squareWithDrawArrays");` function.
- Invokes the `initGL(canvas)` function that obtains the WebGL context using the `canvas.getContext()` function and stores it in a global variable, `gl`. All WebGL API functions are accessed through the context.
- Invokes the `initShaders()` function, which in turn invokes the `getShader(context, type)` function. The `getShader()` function loads both scripts in by parsing the document object. It loads the script in a string object, compiles it, and returns the shader object. Then, the `initShaders()` function creates a program object. It then attaches both the shaders (fragment and vertex) to the program object. Then, the program object is linked to the WebGL context and activated using the context's `useProgram(shaderProgram)` function. Then, the `aVertexPosition` shader attribute provides the location that is requested in the control code using the `gl.getAttribLocation (shaderProgram, "aVertexPosition");` function. This location is stored in the `vertexPositionAttribute` variable of the `shaderProgram` object. This `vertexPositionAttribute` is then activated using the `gl.enableVertexAttribArray()` function. Then the location and both uniforms (ModelView transformation and projection transformation) are stored in the shader program object.
- Invokes the `initBuffers()` function. This function creates the vertices array with four vertices, creates a vertex buffer object, makes it the current buffer using the API function `gl.bindBuffer (gl.ARRAY_BUFFER, vertexPositionBuffer)` , and creates the actual memory to store the

vertices in the GPU memory using the API function `gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW)`. Then, it clears or paints the canvas using the `gl.clearColor(red,green,blue,alpha)` function. It also enables depth-testing (this will be discussed shortly).

- The final drawing happens in the `drawScene()` function. First, it sets the size of the viewport equal to the size of the canvas object using the `gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);` API function. Then, it clears the `COLOR` and `DEPTH` framebuffers (to be discussed shortly). It initializes the projection perspective matrix using the `mat4.perspective(40, gl.viewportWidth / gl.viewportHeight, 0.1, 1000.0, pMatrix)` function.
- The `drawScene()` function then initializes the ModelView matrix `mat4.identity(mvMatrix);` and applies a translation transform to the ModelView matrix. The matrix now contains one transformation which would move an object to $z = -7$ in the world coordinate.
- The `drawScene()` function then makes the `vertexPositionBuffer` parameter the current buffer using the `gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer)` function and associates the current buffer object to the shader attribute using the `gl.vertexAttribPointer()` function. The `setMatrixUniforms()` function associates the ModelView matrix and projection matrix to the corresponding shader program uniforms using the `gl.uniformMatrix4fv()` function.

Finally, we invoke the `drawArrays()` function, which basically draws the square with two triangles using the vertices [v0, v1, v2] and [v2, v0, v3] since the drawing mode was `TRIANGLE_STRIP`.

Hence, for every WebGL application, the basic program flow would be similar to the following:

- Create the WebGL context
- Load, compile, link, and attach shaders
- Initialize vertex buffers and index buffers
- Apply transform and associate buffer objects to shader attributes
- Call the `drawArrays` or `drawElements` functions

Drawing using index buffer objects

Open the `01-SquareWithDrawIndexArrays.html` file in your favorite text editor, which has the following code:

```
function initBuffers() {

  vertexPositionBuffer = gl.createBuffer();
  indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

  vertices = [
              1.0, 1.0, 0.0, //v0
             -1.0, 1.0, 0.0, //v1
              1.0, -1.0, 0.0, //v2
             -1.0, -1.0, 0.0  //v3
  ];
  indices = [0,2,3,0,3,1];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW);
  vertexPositionBuffer.itemSize = 3;
  vertexPositionBuffer.numItems = 4;

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
gl.STATIC_DRAW);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
}
```

In this code, we change the `initBuffer()` function to create index buffers. First, we initialize our array of indices and then create a new buffer of type `gl.ELEMENT_ARRAY_BUFFER,`; we then allocate memory for it using the `bufferData()` function. The process is shown in the following code snippet:

```
function drawScene() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  mat4.perspective(40, gl.viewportWidth / gl.viewportHeight, 0.1,
1000.0, pMatrix);

  mat4.identity(mvMatrix);

  mat4.translate(mvMatrix, [0.0, 0.0, -7.0]);
```

```
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
vertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.drawElements(gl.TRIANGLES, indices.length,
gl.UNSIGNED_SHORT,0);

}
```

In the `drawscene()` function, we make the `indexBuffer` parameter our current `g1.ELEMENT_ARRAY_BUFFER` constant. Then, we use a `drawElements()` function with the `TRIANGLES` mode option.

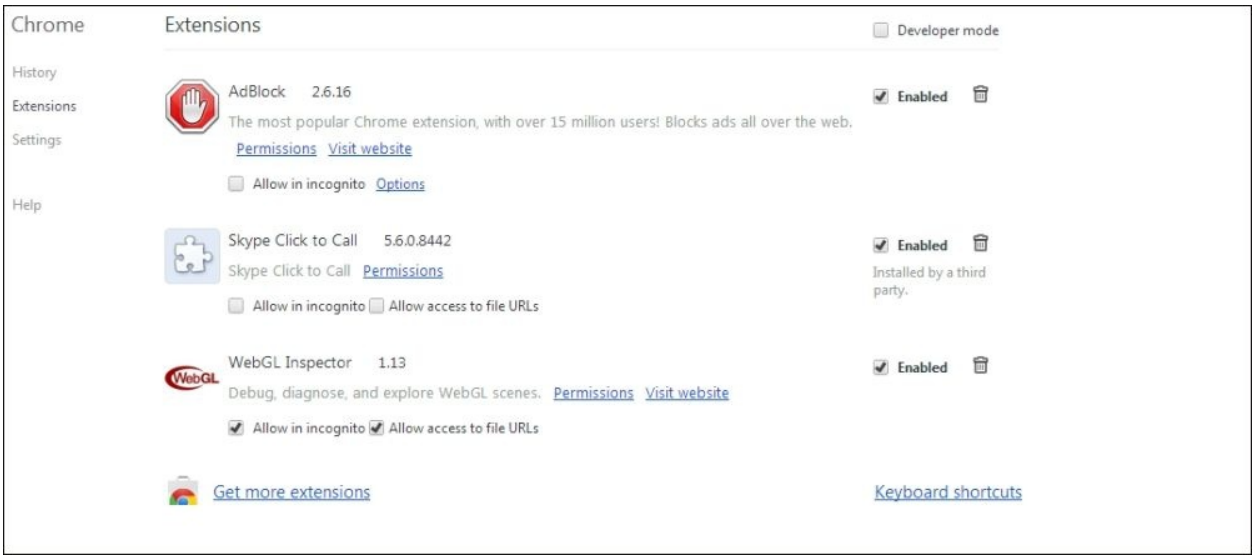
Debugging a WebGL application

A WebGL application is divided into control code and shader code. We can easily debug the control code using a simple JavaScript API, `console.log(canvas)`.

The JavaScript API `console.log()` is not compatible with some browsers, but you can use the wrapper library at https://code.google.com/p/console-js/ for usage across browsers.

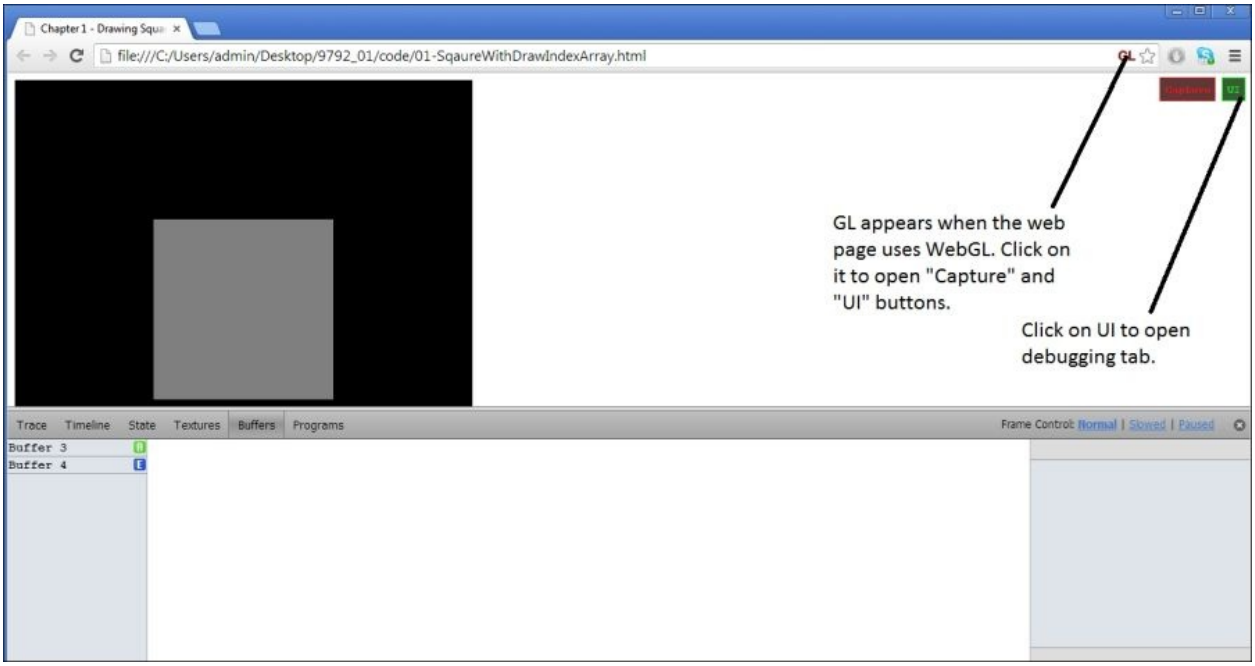
We cannot debug a shader code with a simple JavaScript API. If something goes wrong, all we would see is a black window. Fortunately for us, there is a great Chrome extension named WebGL inspector (http://benvanik.github.io/WebGL-Inspector/) to help us debug our applications.

Simply install the extension in Google Chrome from https://chrome.google.com/webstore/detail/webgl-inspector/ogkcjmbhnfmlnielkjhedpcjomeaghda. We can enable the extension in the **Settings** menu in Google Chrome. Please refer to the following screenshot:



This extension helps debug, diagnose, and explore WebGL games. After enabling it, whenever we render any WebGL scene, a tag GL will appear on the address bar on the right-hand side. When we click on it, two tabs, **Capture** and

UI, appear on the right-hand side of the screen. The **UI** tab opens the debugging window, as shown in the following screenshot:



The debugging window shows us complete information about the buffers, textures, and variable state of the application.

Summary

In this chapter, we got to grips with basic terms such as vertices, indices, vectors, and mesh. We covered a very basic shader and rendered a square using the simple WebGL API. We covered basic variable qualifiers (attribute, uniform, and varying) of the shader language. We covered the basics of loading and compiling a shader program. The core concept of allocating a buffer (vertex buffer objects and index buffer objects) and the processing of attributes in a shader was also covered.

We also discussed components of the WebGL's rendering pipeline, such as vertex shaders, the rasterizer, and fragment shaders.

In the next chapter, we will improve our shader by adding light and material to the objects in the scene.

Chapter 2. Colors and Shading Languages

We have been teleported to 5000 AD. We know the basics, but we still have to create our survival kit to fight Mr. Green. Our survival kit's raw materials will be some basic code packets. These code packets will form the core rendering engine of our WebGL game. Although we will take the simplest approach, we will keep evolving and structuring these particular code packets as we move along.

Before we dive into shading models, we will cover how to render complex geometries. A very important aspect of 3D gaming is the ability to render 3D models exported from tools such as Blender. We will learn how to export basic 3D models and also understand the basic Wavefront OBJ file format. We will learn how to parse the OBJ file format to convert to the JSON format. Then, we will load the JSON file format in our game.

Our survival kit in 5000 AD will cover the following topics:

- Adding colors to primitive objects
- Exporting 3D objects from tools such as Blender
- Understanding directional lights
- Making our objects respond to light and reflection algorithms
- Shading models

Understanding colors

Similar to most graphics systems, WebGL also uses the RGBA model to illuminate pixels on a screen. The alpha channel (A) is the extended attribute to denote the opacity of an object. We will use this value for advanced techniques, such as blending, in the following chapters. Each value of RGBA ranges from `0.0` (none) to `1.0` (full intensity) in WebGL. The value of `1.0` for the alpha channel will denote that the object is opaque and the value of `0.0` will denote that the object is transparent.

We will store color values in the `vec4` data type as follows:

```
var color=
vec4.fromValues(0.0,0.0,1.0,1.0);//(red,green,blue,alpha)
//Blue color with full intensity
var color=vec4.fromValues(1.0,0.0,0.0,1.0);//Red color with full
intensity
```

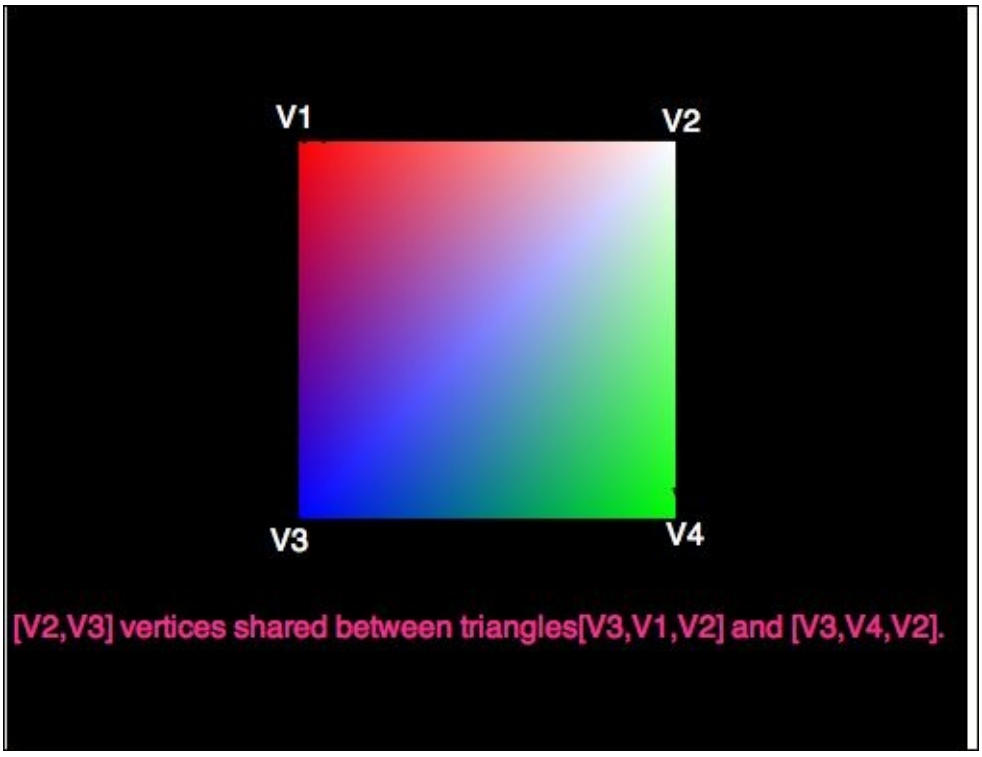
Colors in WebGL are used to represent object material properties, light properties, and painting/clearing of the WebGL context.

Like vertices' positions, color values are stored for shaders in vertex buffer objects.

Coloring our square

Our example uses per-vertex coloring. Per-vertex coloring is rarely used in gaming applications. It is mostly used in engineering applications, but it is a good example to start with as you might run into cases where you need to use per-vertex coloring in games.

In our previous examples, we discussed that each vertex had a position. These positions were stored in flat arrays. In our example of the square, we had four vertices shared between two triangles. Now, we will assign color [R, G, B] values to each vertex so that we get the square colored as shown in the following screenshot:



Coloring using the vertex color

Open the `02-SqaureDrawIndexArrayColor.html` file in your favorite text editor.

The shader code for the vertex color is as follows:

```
<script id="shader-vs" type="x-shader/x-vertex">
```

```
  attribute vec4 aVertexColor;
  //Attribute hold color for each vertex

  varying lowp vec4 vColor;
  //Varying qualifier to pass vertex color  to fragment shader
  void main(void) {

    vColor = aVertexColor; //Assign attribute value to varying to
be used in shader
  }
</script>
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying lowp vec4 vColor;//Varying qualifier to get values from
vertex shader

  void main(void) {

    gl_FragColor = vColor;//Assign value to be used in shader
  }
</script>
```

The preceding code denotes changes to the vertex and fragment shaders used previously in our examples. Three changes in the vertex shader are as follows:

- We create an attribute (`aVertexColor`) to get the vertex color value from the control JavaScript code.
- We create a varying variable to pass this color value to the fragment shader. You might wonder why we did not create a uniform qualifier and pass it directly to the fragment shader from the control code. We will do this in cases where the whole primitive has the same color value. In our case, the primitive is the triangle, and each vertex has a different value. We only use the uniform qualifier for variables whose value does not change across a primitive (across the face of a triangle). Therefore, in our case, we created an attribute which received each value of the vertex from the buffer. Then, we passed this value to the fragment shader using the varying variable. Remember, varying is the only qualifier used to pass values from the vertex shader to the fragment shader. The varying values are interpolated by the hardware, as explained in Chapter 1, *Getting Started with WebGL Game Development*.
- We assign the value of the `aVertexColor` attribute to the varying variable `vColor`.

Two changes in the fragment shader are as follows:

- We create a varying variable, `vColor`, whose value we will receive from the fragment shader.
- We assign the value to the predefined `gl_FragColor` variable.

The control code for the vertex color is as follows:

```
function initBuffers() {
...
...
//  Array of color values, one for each vertex
  var colors = [
    1.0,  1.0,  1.0,  1.0,    // white
    1.0,  0.0,  0.0,  1.0,    // red
    0.0,  1.0,  0.0,  1.0,    // green
    0.0,  0.0,  1.0,  1.0     // blue
  ];
...
...
...
  //New Buffer to hold colors
  colorBuffer = gl.createBuffer();
  //Created reference to the new buffer, made it the active buffer
for further operations
  gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
  //Allocated memory for the buffer and stored color values
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
gl.STATIC_DRAW);
}
```

The preceding code denotes changes to the `initBuffer()` function. In the earlier `initBuffer()` function, we created and allocated a buffer for indices and the vertex positions. In the new `initBuffer()` function, we allocated memory to store the color data as well. Also, note that the buffer type is `ARRAY_BUFFER` to store color values; we use `ELEMENT_ARRAY_BUFFER` to store only indices.

The following code gets the reference to the `aVertexColor` attribute and enables it. The variables are declared in shaders but are initialized in the control flow code. Hence, we obtain a reference to them, as shown in the following code:

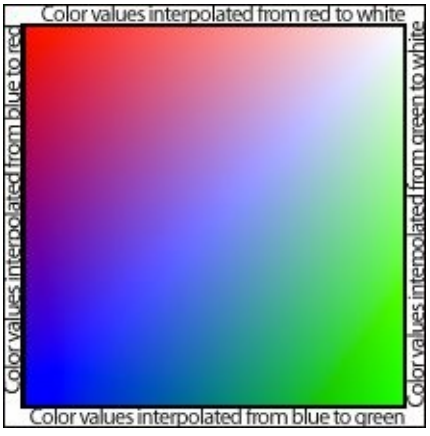
```
function initShaders() {


  shaderProgram.vertexColorAttribute =
```

```
gl.getAttribLocation(shaderProgram, "aVertexColor");// Getting the
reference to the attribute aVertexColor from the flow code
  gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
  //Enabling the vertex color attribute.

}
```

We just added two lines. The first line activates the buffer object and the second line associates the buffer object to the vertex shader attribute `aVertexColor` through the reference `vertexColorAttribute`. The following is the code for the `drawScene()` function:

```
function drawScene() {


  //Activate the buffer object colorBuffer for operations
  gl.bindBuffer(gl.ARRAY_BUFFER,colorBuffer);
  //Associating buffer object(colorBuffer) with shader attribute
aVertexColor whose refrence is in vertexColorAttribute
  gl.vertexAttribPointer(shaderProgram.vertexColorAttribute, 4,
gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.drawElements(gl.TRIANGLES, indices.length,
gl.UNSIGNED_SHORT,0);
}
```

We assigned four color values to each vertex. But, each vertex corresponds to each fragment (pixel on our screen). Hence, the magic happens in the rendering pipeline. We assign a color value to each vertex, and WebGL interpolates it by taking the corresponding calculated color for the vertices surrounding the corresponding fragment (pixel). Hence, you see a gradient from blue to red, green to white, and so on, as shown in the following diagram:


Color values interpolated from red to white
Color values interpolated from blue to red
Color values interpolated from green to white
Color values interpolated from blue to green

The in-between colors are interpolated values by the GPU.

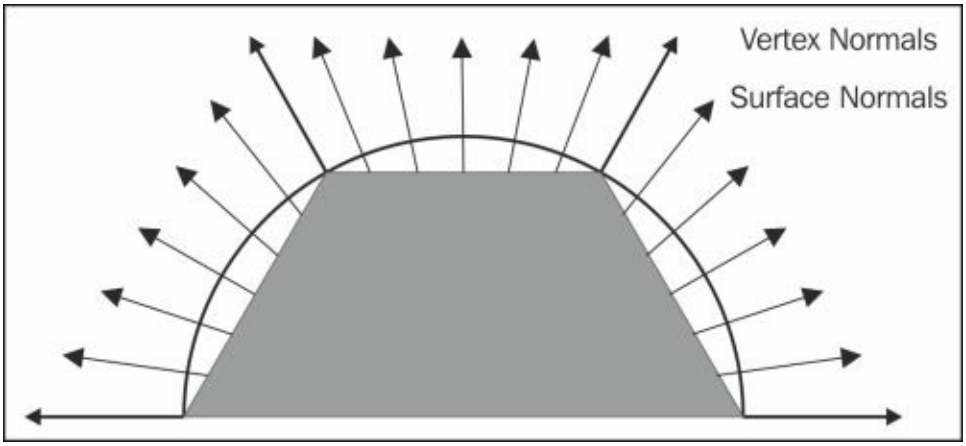
Learning more about colors

In the preceding example, we learned about how to color a simple object by providing a color for each vertex, but in a game, to color an object, we do not generally use vertex colors, but we add a material to an object instead. A material can be modeled by defining parameters such as textures and color. In this chapter, we will model our materials only with color. Material colors are usually modeled as triplets in the **RGB** (**Red**, **Green**, and **Blue**) space.

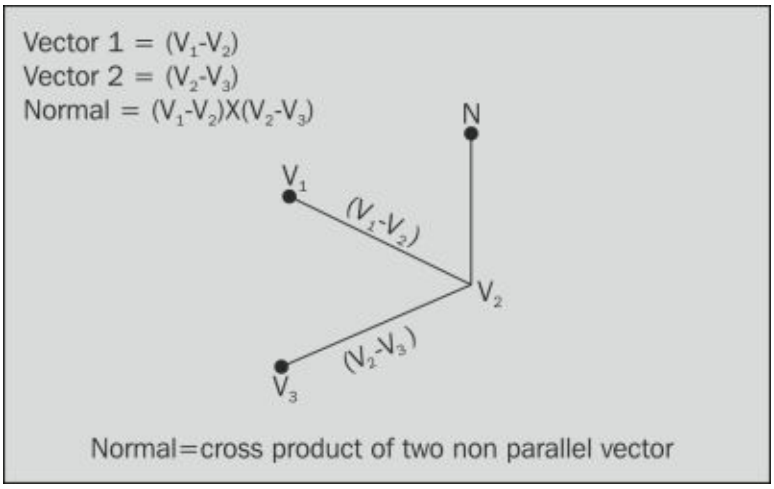
However, the material does not solely decide how the object will appear in your game. The final color of a fragment is defined by how the object is oriented in the scene and lights in the scene. A light is defined by parameters such as direction, position, and colors. Hence, the final color of a fragment is decided by material, orientation (normals), and light intensity.

Understanding surface normals for lighting calculations

A **surface normal** (or just normal) is a vector that is perpendicular to a surface at a particular position. Surface normals and vector normals are explained in the following diagram:



A **normal** is defined as the cross-product of any two nonparallel vectors that are tangent to the surface at a point. We have to have a normal for each vertex of an object. To calculate the normal of a vertex, we will need to do as depicted in the following diagram:



Hence, for the vertex **V2**, its normal would be as follows:

V2 = (**V1 - V2**) x (**V2 - V3**)

The value of the first vector is as follows:

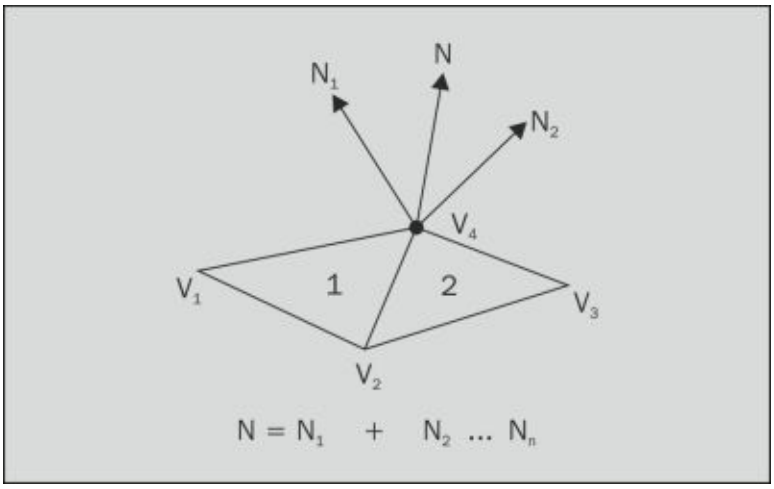
Vector 1 = **V1 - V2**

The value of the second nonparallel vector is as follows:

Vector 2 = **V2 - V3**

V1, **V2**, and **V3** are the three vertices of a triangle primitive.

The calculation of the normal at a vertex shared by multiple triangles/primitives is a little complicated as both primitives contribute to its calculation. The following diagram shows a vertex, **V4**, shared between triangles **1** and **2**:



Hence, if two primitives, (**V1**, **V2**, **V4**) and (**V2**, **V4**, **V3**), share a common vertex, **V4**, then we will calculate **N1** from the primitive **1** (**V1**, **V2**, **V3**) and **N2** from the primitive **2** (**V2**, **V3**, **V4**), and the final normal **N** on vertex **V4** would be the vector sum of **N1** and **N2**.

Calculating normals from vertices and indices

Open the `02-NormalCalculation.html` file in your favorite text editor. A normal is the cross-product of the vectors with common vertices as discussed in the preceding section. The following code calculates the `normals` array from the `vertices` and `indices` arrays:

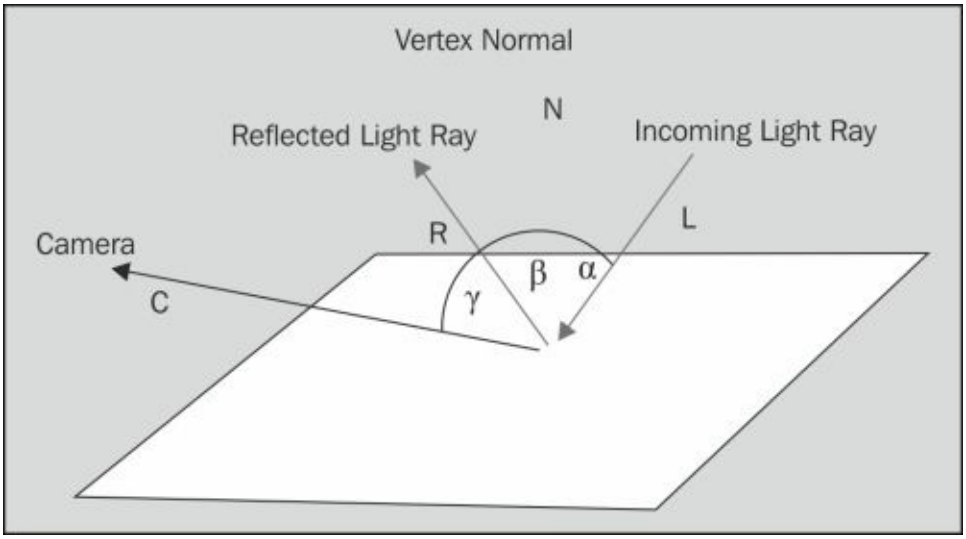
```
function calculateVertexNormals(vertices,indices){
  var vertexVectors=[];
  var normalVectors=[];
  var normals=[];
  for(var i=0;i<vertices.length;i=i+3){
    var
vector=vec3.fromValues(vertices[i],vertices[i+1],vertices[i+2]);
    var normal=vec3.create();//Initialized normal array
    normalVectors.push(normal);
    vertexVectors.push(vector);
  }
  for(var j=0;j<indices.length;j=j+3)//Since we are using triads of
indices to represent one primitive
  {
    //v1-v0
    var vector1=vec3.create();
```

```
vec3.subtract(vector1,vertexVectors[indices[j+1]],vertexVectors[ind
ices[j]]);
    //v2-v1
    var vector2=vec3.create();

vec3.subtract(vector2,vertexVectors[indices[j+2]],vertexVectors[ind
ices[j+1]]);
    var normal=vec3.create();
    //cross-product of two vectors
    vec3.cross(normal, vector1, vector2);
    //Since the normal caculated from three vertices is the same
for all the three vertices(same face/surface), the contribution
from each normal to the corresponding vertex  is the same

vec3.add(normalVectors[indices[j]],normalVectors[indices[j]],normal
);

vec3.add(normalVectors[indices[j+1]],normalVectors[indices[j+1]],no
rmal);

vec3.add(normalVectors[indices[j+2]],normalVectors[indices[j+2]],no
rmal);
  }
  for(var j=0;j<normalVectors.length;j=j+1){
    vec3.normalize(normalVectors[j],normalVectors[j]);
    normals.push(normalVectors[j][0]);
    normals.push(normalVectors[j][1]);
    normals.push(normalVectors[j][2]);

  }
  return normals;
}
```

In the preceding code, the `calculateVertexNormals` function takes the `vertices` and `indices` arrays. The triads of elements of the vertex array define a single vertex; hence, we create a `vec3` object of each triad. Each triad of the `indices` array defines vertices of a triangle; hence, when we iterate over the array, we get all the vertices of the triangle from the `vertexVectors` array. From these vertices, we calculate the normal for that particular vertex. As all three vertices belong to the same face/surface/triangle, the normal of each vertex is the same. Therefore, we calculate only one normal and add it to all the normals of the three vertices. The last loop iterates over the `normalVector` array, normalizes it, and then unpacks the `vec3` object to generate the normal array. We only need the direction of the normal and not its magnitude, since normals are unit vectors.

Understanding normal transformation

The normal is often used in 3D rendering to determine a surface's orientation towards a light source. The following diagram demonstrates the use of the normal to calculate the angle of reflection. We have the direction of light and the surface normal and we need to calculate the direction of the reflected light:



In the upcoming sections, we will discuss how normals are used in reflection/illumination models. These models will be used to calculate color values of the individual fragments of an object.

Like vertices, normals have to be transformed. The normals we calculated in the preceding section are in object space. If the model is translated or rotated, we have to transform the vertices to world space. Similarly, normals have to be transformed to world space as well. To transform normals, we first need to find the transpose of the inverse of the ModelView matrix with the help of the following code:

```
var normalMatrix = mat3.create();
mat4.toInverseMat3(mvMatrix, normalMatrix);
mat3.transpose(normalMatrix);
gl.uniformMatrix3fv(shaderProgram.nMatrixUniform, false,
normalMatrix);//Pass the normal(of the MV matrix) to the shader as
a uniform
```

Then, we multiply the ModelView matrix with the normals of the object in the

vertex shader using the following code:

```
vec3 transformedNormal = nMatrix * aVertexNormal;
```

Different types of lights used in games

In gaming, we add light sources. These light sources can have different lighting effects on our scene and objects. The distinct features of an object only appear if we have added a light to our scene. A scene should have at least one light. The appearance of an object will depend on the direction and position of the light. The following light sources are classified by their distance and direction.

Directional lights

A directional light is produced by a light source placed at an infinite distance from the object/scene. All of the light rays emanating from the light strike the polygons in the scene from a single parallel direction and with equal intensity.

Sunlight is an example of a directional light. It is defined by properties such as color, intensity, and direction.

Point lights

A point light is produced by a light source that gives off equal amounts of light in all directions. Primitives that are closer to the light appear brighter than those that are further away. The intensity varies with distance.

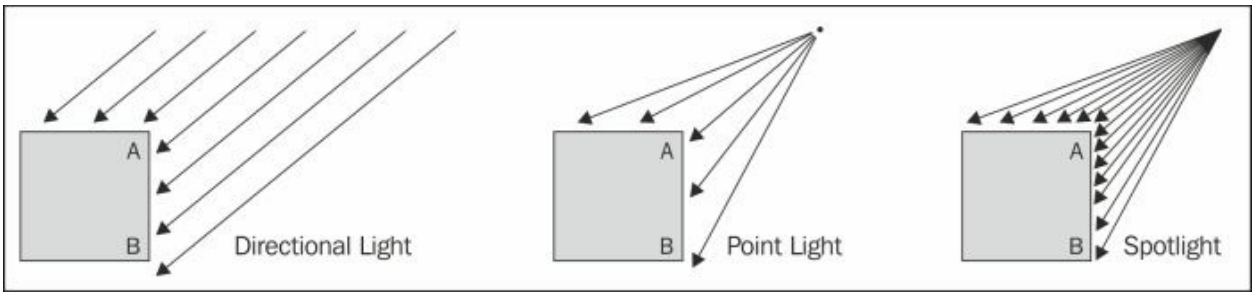
In a game, we will need point lights, similar to the street lamps shown as positional lights in Chapter 3, *Loading the Game Scene*. Point lights are defined by properties such as color, intensity, location, and falloff function.

Spotlights

A spotlight is produced by a light source that radiates light in a cone with maximum intensity at the center, gradually tapering off towards the sides of the cone. The simplest spotlight would just be a point light that is restricted to a certain angle around its primary axis of direction.

A car headlight would be an example of a spotlight. It is defined by properties such as color, intensity, location, an axis of direction, a radius about that axis, and possibly a radial falloff function.

The three light sources are shown in the following diagram:



Ambient lighting

All game scenes have at least one ambient light that produces a uniform illumination. The properties of ambient light are as follows:

- It is generated by a nondirectional light source
- It simulates light that has been reflected so many times from so many surfaces that it appears to come equally from all directions
- Its intensity is constant over a polygon's surface
- The position of the viewer is not important
- Its intensity is not affected by the position or orientation of the polygon in the world
- It is generally used as a base lighting in combination with other lights

The following formula calculates the intensity of light after reflection from an object which has ambient properties:

I = Ia Ka

The parameters of the preceding formula are as follows:

- *I*: Intensity
- *Ia*: The intensity of the ambient light
- *Ka*: The object's ambient reflection coefficient; 0.0 to 1.0 for each of the R, G, and B values

Diffuse reflection (Lambertian reflection)

Diffuse reflection is the reflection of light from a surface such that an incident ray is reflected at many angles rather than just one angle. The properties of diffuse reflection are as follows:

- The brightness of a polygon depends on theta—the angle between the surface normal (N) and the direction of the light source (L)
- The position of the viewer depends on whether we use a directional light or a point light

The following formula calculates the intensity of the reflected diffuse light from an object with a diffuse material component:

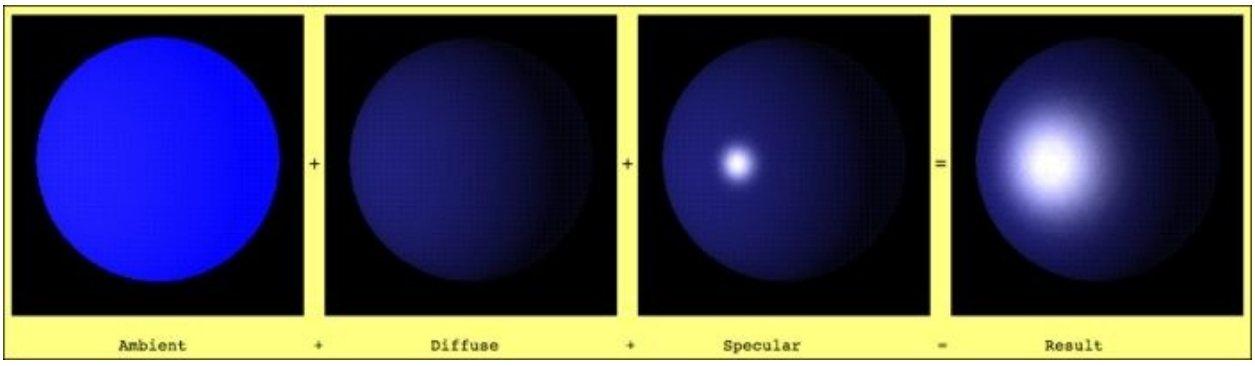
I = Ip Kd (N'.L')

The parameters of the preceding formula are as follows:

- *I*: Intensity
- *Ip*: The intensity of the point light
- *Kd*: The object's diffuse reflection coefficient; 0.0 to 1.0 for each of the R, G, and B values
- *N'*: The normalized surface normal/vertex normal
- *L'*: The normalized direction to the light source
- .: Represents the dot product of the two vectors

Specular reflection

Specular reflection is the reflection of shiny surfaces. You will see a highlight on the object/model. A shiny metal or plastic surface has a high specular component while chalk or carpet has a very low specular component. The following diagram shows the effect of different light components on geometry:



Understanding object materials

A material is a description of the surface of a 3D object. The properties detail a

material's diffuse reflection, ambient reflection, and specular highlight characteristics. The diffuse and ambient properties of the material describe how a material reflects the ambient and diffuse lights in a scene. As most scenes contain much more diffuse light than ambient light, diffuse reflection plays the largest part in determining the color of a primitive. Diffuse and ambient reflections work together to determine the perceived color of an object and are usually identical values. For example, to render Mr. Green, you create a material that reflects only the green component of the diffuse and ambient lights. When placed in a scene with a white light, Mr. Green appears to be green. However, in a room that has only blue light, Mr. Green would appear to be black because its material does not reflect the blue light.

Specular reflection creates highlights on objects making them appear shiny. It is defined by two properties that describe the specular highlight color as well as the material's overall shininess. You establish the color of the specular highlights by setting the specular property to the desired RGBA color—the most common colors are white or light gray. The values you set in the power property control how sharp the specular effects are.

Rendering 3D objects

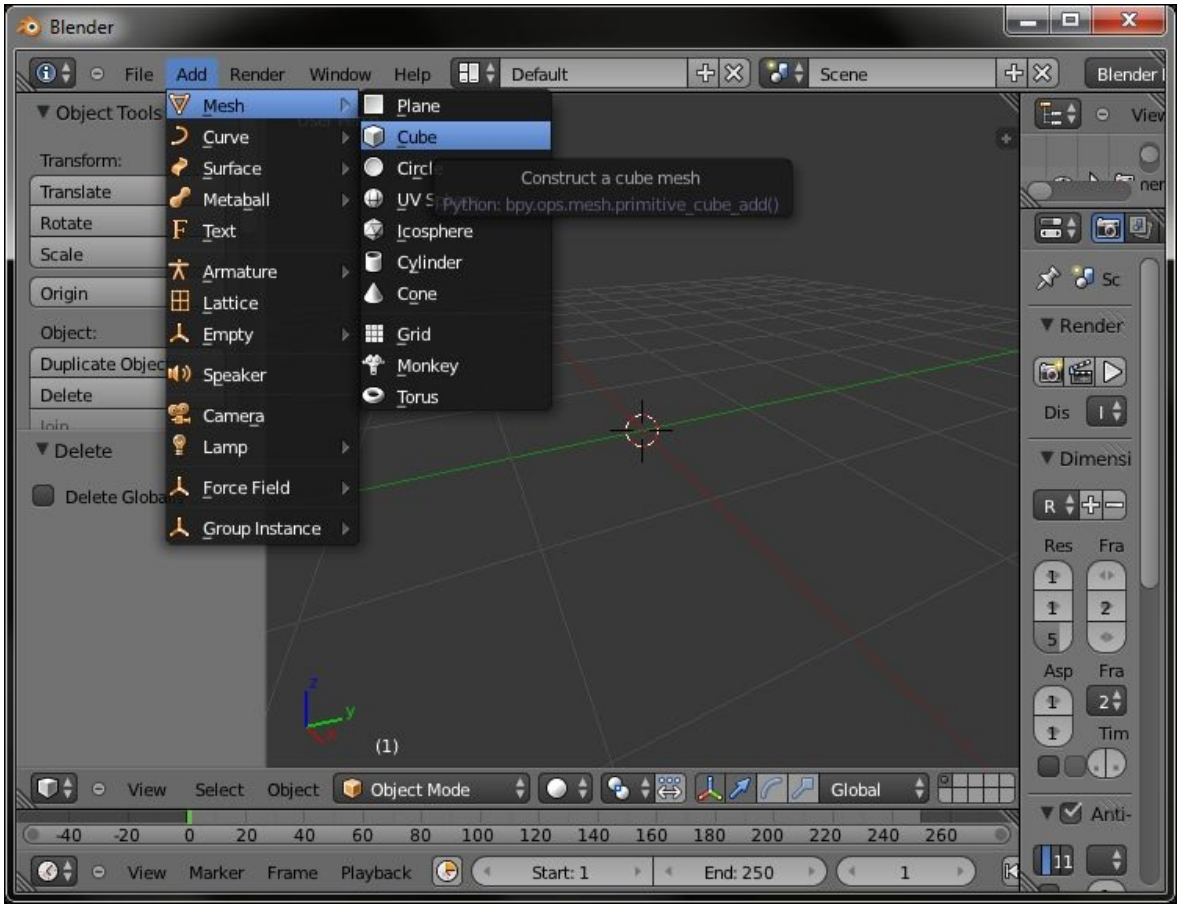
To put our earlier knowledge to use, first we need to learn how to render 3D objects. We will first create a 3D object, a cube or any geometry, in Blender, a free and open source 3D creation software (http://www.blender.org/).

We will then export this 3D model as a Wavefront object file. The Wavefront `.obj` format will then be parsed via a Python script and will be converted and saved as a JSON file. We use the JSON format as it can be easily parsed by JavaScript. We can parse an OBJ file in JavaScript as well but it would be time-consuming and slow.

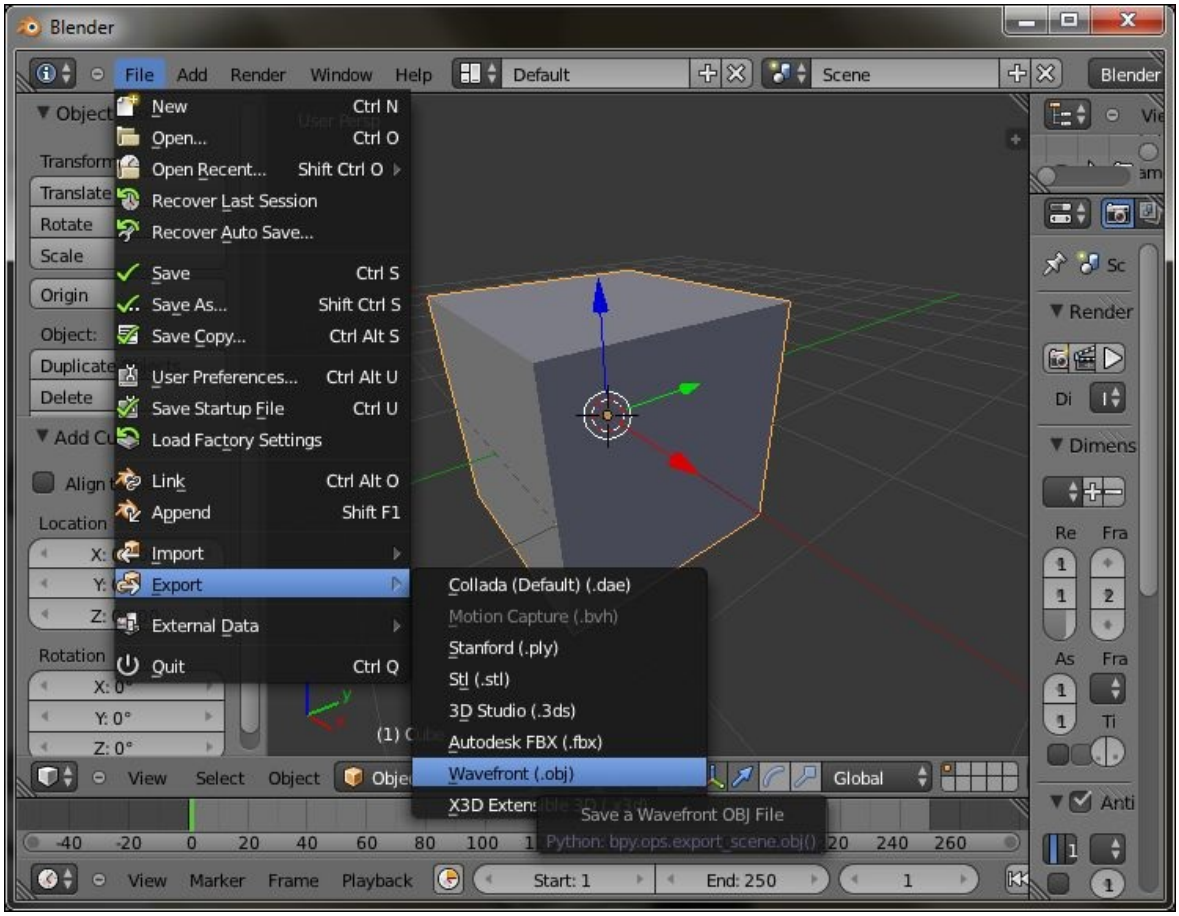
Exporting a 3D object from Blender

The following are the steps we need to perform to export an object from Blender:

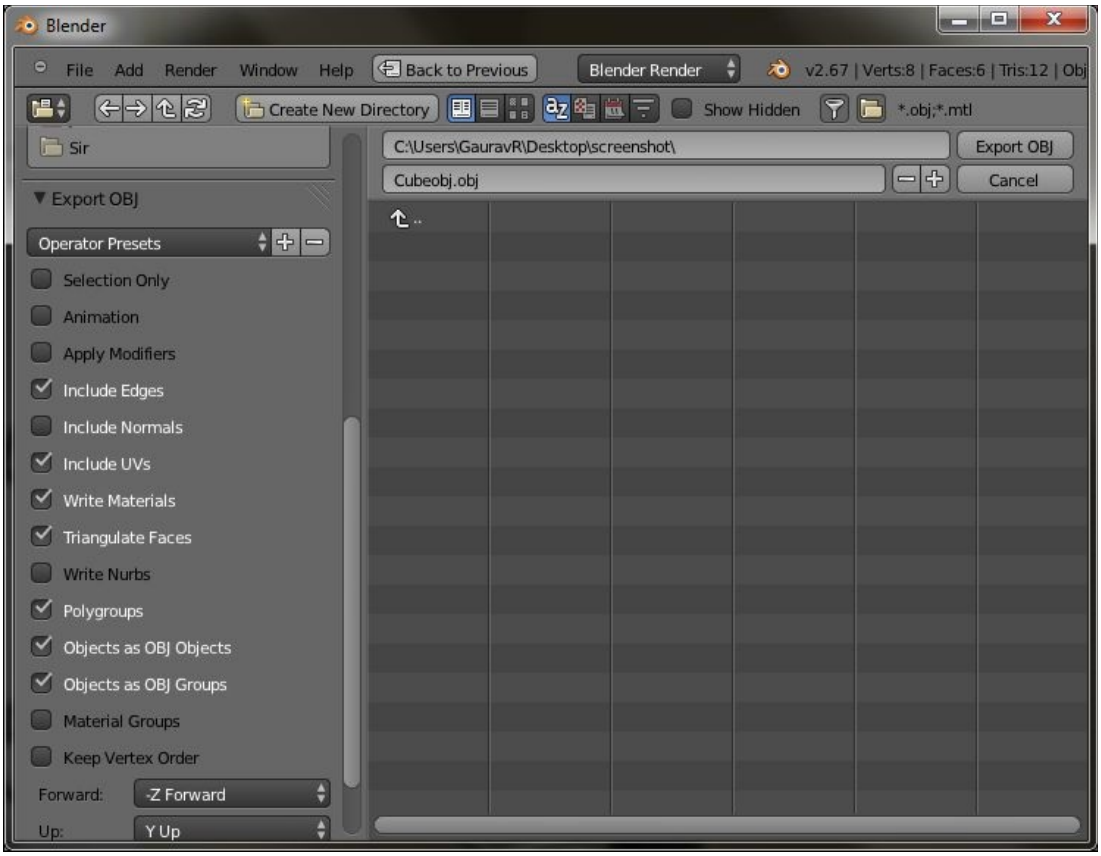
1. Open Blender and go to **Add** | **Mesh** | **Cube**, as shown in the following screenshot:



2. Select **Cube** and go to **File** | **Export** | **Wavefront(.obj)**, as shown in the following screenshot:



- On the left-hand side of the **Export OBJ** window (refer to the following screenshot), select the following options:
  - **Include Edges**
  - **Include UVs**
  - **Write Materials**
  - **Triangulate Faces**
  - **Polygroups**
  - **Objects as OBJ Objects**
  - **Objects as OBJ Groups**



3. Provide a file name and path to save your OBJ file and you are done.

Note

Two files will be generated, one an OBJ file and the other an MTL file. The OBJ file contains the geometry information and the MTL file contains the material information.

Note that we have selected the **Triangulate Faces** option. This will split each polygon into a set of three vertices.

Understanding and loading the Wavefront (OBJ) format

Open the exported `.obj` file in your favorite text editor. The following is the code of an `.obj` file; the inline comments are added to give you a better understanding.

We will only explain the section of the file format relevant to this chapter. We will cover texture data in Chapter 4, *Applying Textures*.

The following code snippet is taken from a `.obj` file of a monkey model exported from Blender:

```
# Blender v2.64 (sub 0) OBJ File: ''
# www.blender.org
# Material File name
mtllib Monkey.mtl
# The object name exported.
o Monkey
#These are vertex information of the above object
v 0.585466 0.897006 0.581131
v -0.289534 0.897006 0.581131
```

Texture coordinates, in (u, v, [w]) format, vary between 0 and 1. The w coordinate is optional. We will cover this later in Chapter 4, *Applying Textures*.

The following code snippet from the `.obj` file lists the vertex normal and the vertex texture data:

```
vt 0.500 1 [0]
...
# Vertex normals information
vn 0.707 0.000 0.707
vn 0.707 0.000 0.707
...
# Start  mapping from the material file; Default None if no
material applied
usemtl [Material name]
# Smoothing is on or off (For Shaders)
s off
#Indices of a surface in our case triangle
f 47 1 3
# We checked to "triangulate" while exporting, hence we have triads
in faces
```

```
...
f 47 3 45
# Vertex/Texture Mapping
f 3/1 4/2 5/3
#Vertex/Texture Mapping/Normal information
f 6/4/1 3/5/3 7/6/5
#Vertex//Normal
f 6//1 3//3 7//5
```

In the preceding file, the key concept to understand is the three data elements, `v`, `vn`, and `vt`:

- `v`: This refers to the vertex indices of an object.
- `vn`: This refers to the vertex normal information.
- `vt`: This holds the texture information. We will not use this data element now. It will be discussed in Chapter 4, *Applying Textures*.

The important element is `f`, faces. A **face** in an OBJ file defines the properties of each polygon. It has indices of the vertices/normals/textures defined in the preceding file. In the preceding example, a face has three vertices as we had selected the option **Triangulate Faces** while exporting the file. If we exported without triangulation, we would have had four vertices per face.

The relation between `v`, `vn`, and `vt` in `f` is shown in the following table:

Cases	Description
Case 1: `f 1 2 12`	The element `f` holds only the vertex information of the face.
Case 2: `f 2/2 3/5 4/5`	The pair maps the vertices to texture coordinates; the second vertex maps to the second texture coordinate.
Case 3: `f 2/2/2 3/2/4 3/3/4`	The value `2/2/2` would mean the second vertex is mapped to the second texture coordinate in the `vt` array and the second element is the normal index in the normal array `vn`.
Case 4: `f`	The value `2//2` would mean the second vertex is mapped to the

Case 4: `f 2//2 3//4 3//4`	The value `2//2` would mean the second vertex is mapped to the second normal in the `v` and `vn` data elements.

Understanding the material file format (MTL)

Open the exported MTL file in your favorite text editor. The following is the code of an MTL file; the inline comments are added to build a better understanding:

```
# Blender MTL File: 'None'
# Material Count: 1
#Material name definition
newmtl [MaterialName]
#Specifies the specular exponent for the current material. This
defines the focus of the specular highlight.
Ns 96.078431
#The ambient color of the material is declared using Ka. Color in
RGB where  value is between 0 and 1.
Ka 0.000000 0.000000 0.000000
#The diffuse color is declared using Kd.
Kd 0.640000 0.640000 0.640000
#The specular color is declared using Ks.
Ks 0.500000 0.500000 0.500000
#Ni optical_density #Specifies the optical density for the surface.
This is also known as index of refraction.
Ni 1.000000
d 1.000000
illum 2
```

The `illum` data element in the `.mtl` file helps us understand the illumination model that needs to be applied when using this material. It can be a number from 0 to 10. The illumination models are summarized as follows:

Value	Description
`0`	Color on and ambient off
`1`	Color on and ambient on
`2`	Highlight on

3	Reflection on and ray trace on
4	Transparency: glass on; reflection: ray trace on
5	Reflection: fresnel on and ray trace on
6	Transparency: refraction on; reflection: fresnel off and ray trace on
7	Transparency: refraction on; reflection: fresnel on and ray trace on
8	Reflection on and ray trace off
9	Transparency: glass on; reflection: ray trace off
10	Casts shadows onto invisible surfaces

Converting the OBJ file to the JSON file format

The JSON file format does not have any particular specification in WebGL. We store our object data in JSON because it is the easiest way to parse in JavaScript. Although we can write our own script to create a JSON file from the OBJ file, we would prefer to use an existing script to do the same:

- Install Python 3.x from https://www.python.org/downloads/. (Macintosh generally has Python installed.)
- Download the script `convert_obj_three.py` from https://github.com/mrdoob/three.js/blob/master/utils/converters/obj/convert_ (The script is already attached with the chapter code files.)

Use exported files from Blender, `objectname.obj` and `objectname.mtl`, in the following command.

If you are using a Mac, open Terminal, or if you are using Windows, open Command Prompt and enter the following command:

```
python  convert_obj_three.py -i objectname.obj -o objectname.json
```

The JSON file

Open the `Cube.json` file from the `model` folder in your favorite text editor. The following code is of the `Cube.json` file:

```
{
"metadata" :
  {
    "formatVersion" : 3.1,
    "sourceFile"    : "Cube.obj",
    "generatedBy"   : "OBJConverter",
    "vertices"      : 8,
    "faces"         : 12,
    "normals"       : 0,
    "colors"        : 0,
    "uvs"           : 0,
    "materials"     : 1
  },
  "scale" : 1.000000,
  "materials": [{
    "DbgColor" : 15658734,
    "DbgIndex" : 0,
    "DbgName" :"",
    "colorAmbient" : [0.0, 0.0, 0.0],
    "colorDiffuse" : [0.80000000000000004, 0.80000000000000004,
0.80000000000000004],
    "colorSpecular" : [0.80000000000000004, 0.80000000000000004,
0.80000000000000004],
    "illumination" : 2,
    "specularCoef" : 0.0,
    "transparency" : 1.0
  }],
  "vertices": [-0.446862,0.067651,0.815504,........],
  "morphTargets": [],
  "morphColors": [],
  "normals": [],
  "colors": [],
  "uvs": [[]],
  "faces":
[2,1,0,4,0,2,1,4,5,0,2,5,6,2,0,2,5,2,1,0,2,6,7,3,0,2,6,3,2,0,2,0,3,
7,0,2,0,7,4,0,2,0,1,2,0,2,0,2,3,0,2,7,6,5,0,2,7,5,4,0]
}
```

The JSON file contains the complete information of the object. It contains

information of the materials as well as the `vertices` and `faces` arrays. Each triad of elements in the `vertices` array will form a face. Also note that it supplies information on the number of materials used to render the object. In our case, it is `''materials'' : 1` in the metadata section. However, the `"materials": []` object in the JSON file is an array. In the preceding example, this array has a single element.

Understanding the `faces` array is pretty exciting. First, note that there are arrays containing UV, color, and the normal information array as well. As the name `faces` denotes, it stores information on the faces of the polygon. A face will have information on its vertices and the primitive can be a quad or a triangle. The `faces` array can hold information such as a normal index denoting the face normal, material index, and color, as well as UV indices. All indices point to their corresponding arrays. So, a typical face set would be something like the following code snippet:

```
v1,v2,v3,[v4],[material_index],[face_uv],[face_vertex_uv,
face_vertex_uv,face_vertex_uv, face_vertex_uv],[face_normal],
[face_vertex_normal, face_vertex_normal,face_vertex_normal,
face_vertex_normal],[face_color],[face_vertex_color,
face_vertex_color,face_vertex_color, face_vertex_color],
```

The square brackets `[]` denote an optional element. Hence, from the preceding string, we understand that a face information set may or may not have all this information. So, how do we know whether our face set has all this information?

We learn about our face set after reading the first element of the array. The value of the first element of the array will range from 1 to 255. We will first convert this number to an octet. Each bit of the octet has information about the face. The combination of bits helps us understand the number of elements we have to read to get the information for that face, as demonstrated in the following code snippet:

```
00 00 00 00 = TRIANGLE
00 00 00 01 = QUAD
00 00 00 10 = FACE_MATERIAL
00 00 01 00 = FACE_UV
00 00 10 00 = FACE_VERTEX_UV
00 01 00 00 = FACE_NORMAL
00 10 00 00 = FACE_VERTEX_NORMAL
01 00 00 00 = FACE_COLOR
10 00 00 00 = FACE_VERTEX_COLOR
```

The first element can be one of the elements present in the following table:

Value	Binary format	Description
0	00000000	This face is a triangle, and the next three values will be its vertices.
1	00000001	This is a quad. Hence, the next four elements will be vertices of a quad.
2	00000010	As the last bit is 0, it is a triangle, and the second bit is 1 which means it will face the material index. The first three values will denote the vertex indices (of the earlier vertex array), and the next value will give the material index of the material array (`v1`, `v2`, `v3`, `materialIndex`). Each element of the material array has information (specular, diffuse, ambient, and so on) of the material used on the object.
3	00000011	As the last bit is 1, it is a quad, and the second bit is 1 which means it will face the material index. Hence, we will read the next five elements (`v1`, `v2`, `v3`, `v4`, `materialIndex`).
4	00000100	As the last bit is 0, it is a triangle. The second bit is 0 (no material) and the third bit is 1, which means it has a face UV. Hence, we will read the next four elements (`v1`, `v2`, `v3`, `faceUVIndex`).

The following code snippet is taken from a `.json` file. It shows different data elements and their corresponding meanings:

```
"faces": [
  // triangle
  // 00 00 00 00 = 0
  // 0, [vertex_index, vertex_index, vertex_index]
  0, 0,1,2,
  // quad
```

```
  // quad
  // 00 00 00 01 = 1
  // 1, [vertex_index, vertex_index, vertex_index, vertex_index]
  1, 0,1,2,3,
  // triangle with material
  // 00 00 00 10 = 2
  // 2, [vertex_index, vertex_index, vertex_index],
  // [material_index]
  2, 0,1,2, 0,
  // triangle with material, vertex uvs and face normal
  // 00 10 01 10 = 38
  // 38, [vertex_index, vertex_index, vertex_index],
  // [material_index], [vertex_uv, vertex_uv, vertex_uv],
  // [face_normal]
  38, 0,1,2, 0, 0,1,2, 0,
  // triangle with material, vertex uvs and vertex normals
  // 00 10 10 10 = 42
  // 42, [vertex_index, vertex_index, vertex_index],
  // [material_index],[vertex_uv, vertex_uv, vertex_uv],
  // [vertex_normal, vertex_normal, vertex_normal]
  42, 0,1,2, 0, 0,1,2, 0,1,2,
  // quad with everything
  // 11 11 11 11 = 255
  // 255, [vertex_index, vertex_index, vertex_index, vertex_index],
  //  [material_index],[face_uv],
  //[face_vertex_uv, face_vertex_uv, face_vertex_uv,
face_vertex_uv],
  //  [face_normal],[face_vertex_normal, face_vertex_normal,
  //   face_vertex_normal, face_vertex_normal],[face_color]
  //  [face_vertex_color, face_vertex_color,
  //   face_vertex_color, face_vertex_color],
  255, 0,1,2,3, 0, 0, 0,1,2,3, 0, 0,1,2,3, 0, 0,1,2,3,
]
```

Parsing the JSON faces array

Open the `parseJSON.js` file from the `primitive` folder in your favorite editor. The following function checks whether the bit position is set in an integer:

```
function isBitSet( value, position ) {
  return value & ( 1 << position );
}
```

The following code reads the first element of the array, checks each bit, and stores the Boolean value of the status in the corresponding variable:

```
function parseJSON(data){
  var faceArray=[];
```

```
  var faceArray=[];
  var i, j, fi,offset, zLength, nVertices,colorIndex, normalIndex,
uvIndex, materialIndex,type,isQuad,hasMaterial,hasFaceUv,
hasFaceVertexUv,hasFaceNormal, hasFaceVertexNormal, hasFaceColor,
hasFaceVertexColor,vertex, face, color, normal, uvLayer, uvs, u, v,
  faces = data.faces,
  vertices = data.vertices,
  normals = data.normals,
  colors = data.colors,
  nUvLayers = 0;
  // Count the number of UV elements
  for ( i = 0; i < data.uvs.length; i++ ) {
    if ( data.uvs[ i ].length ) nUvLayers ++;
  }
  offset = 0;
  zLength = faces.length;
  while ( offset < zLength ) {
    type = faces[ offset ++ ];
    isQuad              = isBitSet( type, 0 );
    hasMaterial         = isBitSet( type, 1 );
    hasFaceUv           = isBitSet( type, 2 );
    hasFaceVertexUv     = isBitSet( type, 3 );
    hasFaceNormal       = isBitSet( type, 4 );
    hasFaceVertexNormal = isBitSet( type, 5 );
    hasFaceColor        = isBitSet( type, 6 );
    hasFaceVertexColor  = isBitSet( type, 7 );
```

The following code checks whether the first bit is 0 and then reads the next four values. If it is a triangle, it reads the next three values. For each face definition in the `faces` array, we create a JavaScript object. The object's predefined properties such as vertices, `materialIndex`, and `normalIndex` are then initialized with the data from the array:

```
if ( isQuad ) {
  face = new Face();
  face.a = faces[ offset++ ];
  face.b = faces[ offset++ ];
  face.c = faces[ offset++ ];
  face.d=faces[ offset++ ];
  nVertices = 4;
} else {
  face = new Face();
  face.a = faces[ offset++ ];
  face.b = faces[ offset++ ];
  face.c = faces[ offset++ ];
  nVertices = 3;
}
```

The following code checks for the material and increments the offset:

```
if ( hasMaterial ) {
  materialIndex = faces[ offset++ ];
  face.materialIndex = materialIndex;
}
```

The following code checks whether it has face normals and then reads and stores the index of the normal array. Every face will have a normal:

```
//Just iterating and moving offset index forward. UV not relevant
to this chapter.
if ( hasFaceUv ) {
  for ( i = 0; i < nUvLayers; i++ ) {

    uvIndex = faces[ offset++ ];
  }
}
//Just iterating and moving offset index forward. UV not relevant
to this chapter.
if ( hasFaceVertexUv ) {
  for ( i = 0; i < nUvLayers; i++ ) {
    for ( j = 0; j < nVertices; j++ ) {

      uvIndex = faces[ offset++ ];
    }
  }

}
if ( hasFaceNormal ) {
  normalIndex = faces[ offset++ ] * 3;
  normal = vec3.fromValues(normals[ normalIndex++ ],normals[
normalIndex++ ],normals[ normalIndex ]);
  face.normal = normal;

}
```

The following code checks whether it has vertex normals and then reads and stores the normal array indices for each vertex of the face. Every vertex of a face will have a normal:

```
if ( hasFaceVertexNormal ) {
  for ( i = 0; i < nVertices; i++ ) {
    normalIndex = faces[ offset++ ] * 3;
    normal = vec3.fromValues(normals[ normalIndex++ ],normals[
normalIndex++ ],normals[ normalIndex ]);
```

```
normalIndex++ ],normals[ normalIndex ]);
    face.vertexNormals.push( normal );
  }
}
```

The following code checks whether it has the face color and then reads and stores the index of the color array. Each face will have a color:

```
if ( hasFaceColor ) {

  colorIndex = faces[ offset++ ];

  face.colorIndex = colorIndex;

}
```

The following code checks whether it has vertex colors and then reads and stores the color array's indices for each vertex of the face. Every vertex of the face will have a color:

```
if ( hasFaceVertexColor ) {
  for ( i = 0; i < nVertices; i++ ) {
    colorIndex = faces[ offset++ ];
    face.vertexColors.push( colorIndex );
  }
}
faceArray.push( face );
}
```

The following function, `getIndicesFromFaces()`, iterates over the `faces` array and populates the `indices` array:

```
return faceArray;
}
function getIndicesFromFaces(faces){
var indices=[];
for(var i=0;i<faces.length;++i){
indices.push(faces[i].a);
indices.push(faces[i].b);
indices.push(faces[i].c);

    }
return indices;
}
```

Note

This JSON array format discussed here is not an industry standard. This format was created by developers of three.js (a WebGL library). This JSON format is capable of storing objects as well as the complete scene.

Loading the JSON model

Open the `02-Loading-Model-JSON.html` file in your favorite editor.

The vertex shader remains the same as in the example `02-SqaureDrawIndexArrayColor.html`.

The following code defines a simple fragment shader. We have added a `vec4` variable, `materialDiffuseColor`, and qualified it as `uniform`. We pass the value of the variable from the control.

The changes in the fragment shader code to load the JSON model are as follows.

Note that in the previous example (`02-SqaureDrawIndexArrayColor.html`), we passed the color as an attribute and then passed it as a `varying` qualifier to the fragment shader. But in this example, we pass the color as a `uniform` qualifier, as shown in the following code:

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  uniform vec4 materialDiffuseColor;
  void main(void) {
    gl_FragColor = materialDiffuseColor;
  }
</script>
```

We did so because the color of each vertex was different; we pass values as an attribute when the property of each vertex is different for the same primitive. If the value of any vertex property does not change for a primitive, then we pass that value as `uniform`. In this particular example, the color of each vertex remains the same. The object is shaded with the same diffuse color for all vertices.

The changes in the control code to load the JSON model are as follows:

```
function start() {
  var canvas = document.getElementById("squareWithDrawArrays");
  initGL(canvas)
  initShaders();
```

```
  loadModel();
  document.onkeydown = handleKeyDown;
}
function loadModel(){
  $.getJSON("model/Cube.json",function(data){

    vertices = data.vertices;
    var faces=parseJSON(data);
    indices = getIndicesFromFaces(faces);
    if(data.materials.length>0){
      diffuseColor=data.materials[0].colorDiffuse;
      diffuseColor.push(1.0);//Added alpha channel

    }

    initScene();
  });
}
function initScene(){
  initBuffers();
  gl.clearColor(0.6, 0.6, 0.6, 1.0);
  gl.enable(gl.DEPTH_TEST);
  drawScene();
}
```

The `start()` function invokes the `loadModel()` function. The `loadModel()` function invokes the jQuery library's `$.getJSON()` function. The jQuery response handler converts the response string to a JSON object. From the JSON object, we directly assign vertices to the `vertices` array and faces to our `indices` array. Then, we check the length of the `materials` array (as explained earlier, an object can have multiple materials, but in our case, we have a single material) and then assign the `colorDiffuse` variable from the first material object to the `diffuseColor` global variable. Then, the `start()` function invokes the `initScene()` function that initializes the vertex buffer and the index buffer.

```
function initShaders() {
...
  shaderProgram.materialDiffuseColor =
gl.getUniformLocation(shaderProgram, "materialDiffuseColor");
...
}
```

In the preceding code, in the `initShaders()` function, we added a new variable, `shaderProgram.materialDiffuseColor`, that holds the reference to the

`materialDiffuseColor` uniform in the fragment shader.

```
function setMatrixUniforms() {
...


gl.uniform4f(shaderProgram.materialDiffuseColor,diffuseColor[0],dif
fuseColor[1],diffuseColor[2],diffuseColor[3]);
}
```

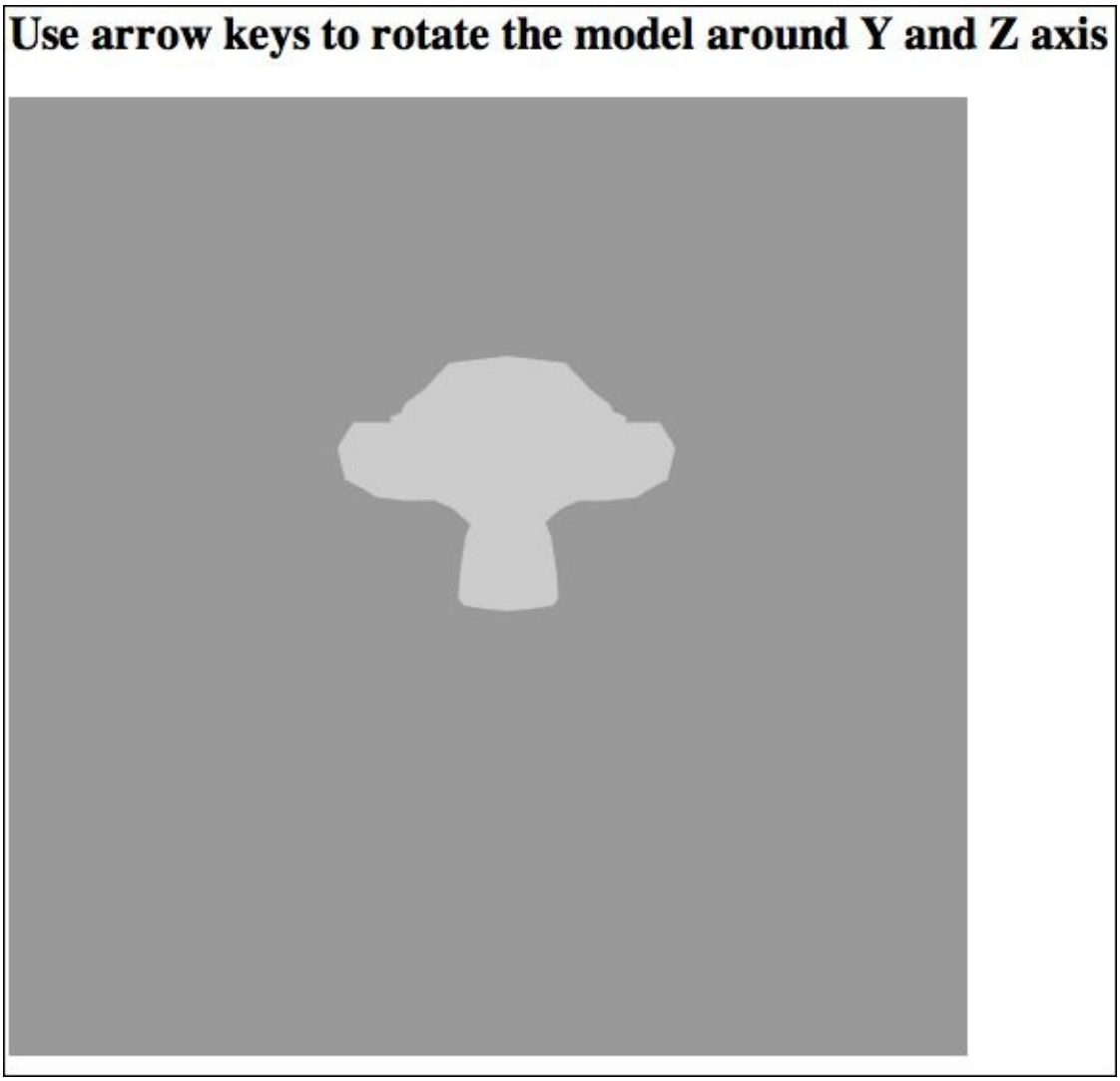
In the `setMatrixUniforms()` function, we associate the values of the `diffuseColor` components to the reference of the `materialDiffuseColor` uniform using the `uniform4f()` function.

```
function drawScene() {
...
...
  mat4.rotateY(mvMatrix,rotateY);
  mat4.rotateZ(mvMatrix,rotateZ);
...
}
```

The `drawScene()` function remains nearly the same, except we have added two functions to add rotation transformation to the ModelView matrix by angles, `rotateY` and `rotateZ`, along the *y* and *z* axes. The values of variables are controlled by the `handleKeys(e)` function, which is invoked when any of the arrow keys are pressed.

Rendering without light

Open the `02-Loading-Model-JSON.html` file in your browser and you will see an outcome similar to the following screenshot. The render does not show distinct object features. We exported a monkey model from Blender in an OBJ file and converted it to JSON. Select **Monkey** from the **Select Model** option. The output of the `02-Loading-Model-JSON.html` file is shown in the following screenshot:



The preceding screenshot certainly does not look like a monkey. Why? Because we did not add any light, illumination, or a shading model to our scene. If these effects are not added, the effect of 3D realism on a 2D screen cannot be achieved.

Making a scene realistic depends on the use of light effects. Do not forget to note that the number of lights and the choice of shading algorithms impact the game performance. We will discuss a few illuminations and shading models, but we will restrict ourselves from using the most used model for gaming. Also, in games, we add textures baked in lighting so that the number of lights used is minimal (we mostly avoid specular reflections). We will cover these textures in

Chapter 4, *Applying Textures*.

Understanding the illumination/reflection model

The appearance of an object depends on the direction in which the light is reflected. This also depends on the changes in the color and intensity of light after reflection. The algorithms used to calculate the direction, intensity, and color of reflected light are called reflection models. Reflection models use the color components (ambient, diffuse, and specular) of a light source as well as the object material to calculate the color of a fragment. Although we will cover these components in depth in subsequent chapters, let us quickly get a basic understanding of how each component contributes to the final color calculation of a fragment.

The **ambient** component illuminates every object equally and is reflected in all directions equally. It simply means that the direction and distance of the light are not used to calculate the ambient color component.

The **diffuse** component is directional in nature. The final diffuse color calculation involves the distance of the light source and the angle the light direction subtends with the surface normal.

The **specular** component is used to highlight an area of the object (shininess caused by reflection). It is also calculated using the direction and distance of the light source.

Rendering images with more realistic reflection models is called **Bidirectional Reflectance Distribution Functions** (**BRDFs**). A BRDF model for computing the reflection from a surface takes into account the input direction of the incoming light and the outgoing direction of the reflected light. The elevation and azimuth angles of these direction vectors are used to compute the relative amount of light reflected in the outgoing direction. BRDF models are constantly evolving to give more realistic effects.

The following are some common BRDF models:

- **The Lambertian model**: This perfectly represents diffuse (matte) surfaces with a constant BRDF.
- **The Phong reflectance model**: This is a phenomenological model akin to

plastic-like specularity.

- **The Blinn-Phong model**: This resembles Phong but allows certain quantities to be interpolated and reduces computational overhead.
- **The Torrance-Sparrow model**: This is a general model that represents surfaces as distributions of perfect specular microfacets.
- **The Cook-Torrance model**: This is a specular-microfacet model (Torrance-Sparrow) that accounts for wavelength and thus color shifting.
- **The Ward's anisotropic model**: This is a specular-microfacet model with an elliptical-Gaussian distribution function dependent on the surface tangent orientation (in addition to the surface normal).

We will cover the Lambertian (for diffuse light/materials) and Blinn-Phong models for specular effect. The reason they are most favored is because of lesser computational overhead.

Lambertian reflectance/diffuse reflection

Lambertian reflectance is the property that defines an ideal diffusely reflecting surface. The apparent brightness of such a surface to an observer is the same regardless of the observer's angle of view. The luminous intensity obeys Lambert's cosine law.

This technique causes all closed polygons (such as a triangle within a 3D mesh) to reflect light equally in all directions when rendered. In effect, a point rotated around its normal vector will not change the way it reflects light. However, the point will change the way it reflects light if it is tilted away from its initial normal vector. The reflection is calculated by taking the dot product of the surface's normal vector and a normalized light-direction vector pointing from the surface to the light source. This number is then multiplied by the color of the surface and the intensity of the light hitting the surface, as shown in the following formula:

$$I_D = -L.NCI_L$$

Here, I^D is the intensity of the diffusely reflected light (surface brightness), C is the color, and I^L is the intensity of the incoming light. $-L.N$ is calculated as follows:

$-L.N = |-L|\,|N| \cos\alpha = \cos\alpha$

Here, $|-L|$ and $|N|$ are equal to 1, which are unit vectors, and α is the angle between the direction of the two vectors. The intensity will be highest if the normal vector points in the same direction as the light vector $\cos(0)=1$ (the surface will be perpendicular to the direction of the light), and lowest if the normal vector is perpendicular to the light vector $\cos(\pi/2)=1$ (the surface runs parallel with the direction of the light). Also, N can be the normal vector or the face vector depending on the shading model used.

The following is the formula to calculate C (color):

$C=C_L C_M$

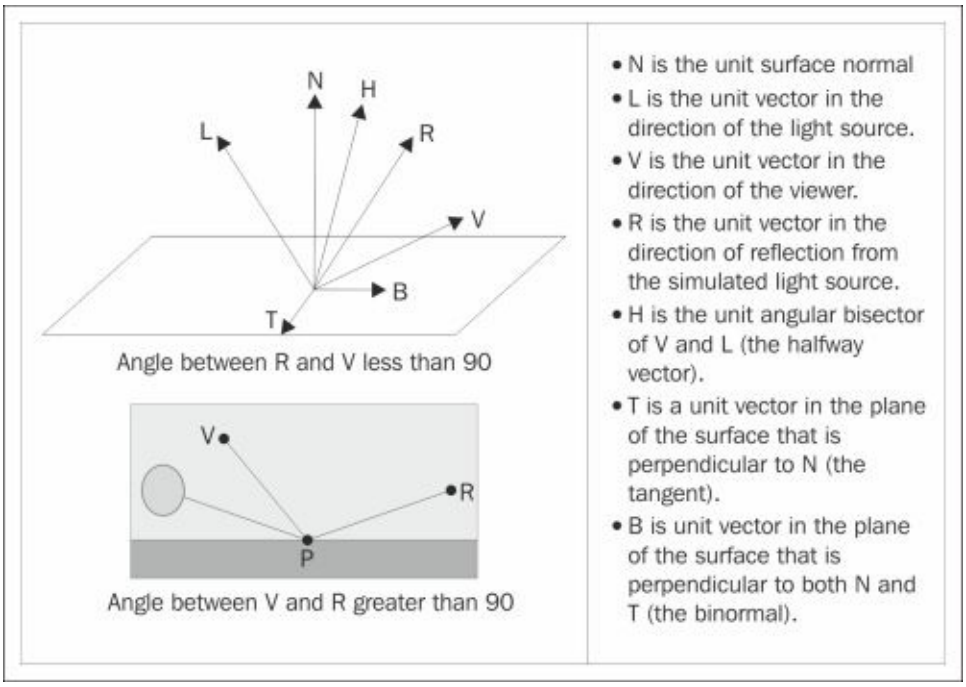
Here, C_L is the color of light and C_M is the color of material.

The code for the Lambert reflectance model is as follows:

```
//Transform the Normal Vertex/Face Vector with normal
matrix(transpose of inverse of mVMatrix
vec3 N = normalize(vec3(uNMatrix  vec4(aVertexNormal, 1.0)));
//Normalize light
vec3 L = normalize(uLightDirection);
//Lambert's cosine law
float lambertTerm = dot(N,-L);
//Final Color
vec4 colorDiffuse = uMaterialDiffuse  uLightDiffuse * lambertTerm;
vFinalColor = colorDiffuse;
vFinalColor.a = 1.0;
```

The Blinn-Phong model

In diffuse reflections, light is reflected in all directions. However, in specular reflection/highlights, the angle of view is important. There are many models used for specular reflection calculations, and amongst them, the most popular is the Phong model. However, the Phong model has a shortcoming as it does not work when the angle between **V** (eye vector) and **R** (light reflection vector) is greater than 90 degrees, as shown in the following diagram:


N
H
L
R
V
B
T
Angle between R and V less than 90
V
R
P
Angle between V and R greater than 90
• N is the unit surface normal
• L is the unit vector in the direction of the light source.
• V is the unit vector in the direction of the viewer.
• R is the unit vector in the direction of reflection from the simulated light source.
• H is the unit angular bisector of V and L (the halfway vector).
• T is a unit vector in the plane of the surface that is perpendicular to N (the tangent).
• B is unit vector in the plane of the surface that is perpendicular to both N and T (the binormal).

The Blinn-Phong shading model (also called the Blinn-Phong reflection model or the modified Phong reflection model) is a modification to the Phong reflection model. In the Blinn model, the model is required to compute the half vector. The half-angle vector (**H**) is the direction halfway between the view direction and the light position. The Blinn model compares the half-angle vector to the surface normal. It then raises this value to a power representing the shininess of the surface:

$H = (L + V)/|L + V|$

Blinn term$= (H.N)^s$

Here, *s* is the shininess of the surface.

The code for the Blinn-Phong model is as follows:

```
vec3 halfDir = normalize(lightDir + viewDir);
float specAngle = max(dot(halfDir, normal), 0.0);
specular = pow(specAngle, 16.0);
vFinalColor = vec4(ambientColor + lambertian * diffuseColor +
specular * specColor, 1.0);
```

The preceding code describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces.

Understanding shading/interpolation models

We discussed earlier that information about the surface of a model, such as the positions of points on the surface and the normal vectors at those points, are stored only for each vertex of a triangle mesh. When a single triangle is rendered, information known at each vertex is interpolated across the face of the triangle. Sometimes we calculate the diffuse and specular illumination only at the vertices of a mesh, and at other times we apply the entire illumination formula at every individual pixel drawn in the display. A shading model defines whether we will apply the illumination model only for the vertex or apply the illumination model for every pixel of the face. If calculated at the vertex level, then the values of the pixels between vertices are linearly interpolated. Remember that if we want to apply the illumination model per vertex, then we will have to compute it in the vertex shader. The vertex shader is executed for each vertex. Hence, it is computationally expensive. Therefore, we would prefer to apply it per pixel in the fragment shader.

The three most common shading models are as follows:

- Flat shading
- Gouraud shading
- Phong shading

Flat shading

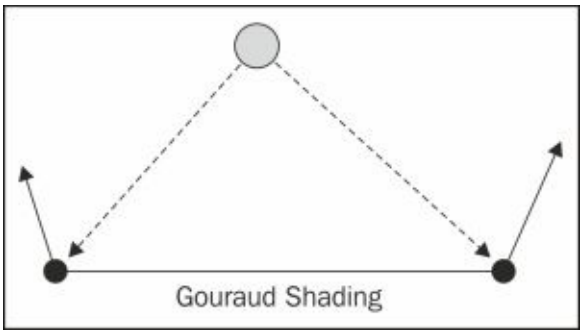
Flat shading shades each polygon of an object based on the angle between the face surface normal and the direction of the light source, its respective colors, and the intensity of the light source. It is usually used for high-speed rendering where more advanced shading techniques are too computationally expensive.

Flat shading gives low-polygon models a faceted look. We do not use flat shading for gaming applications.

Gouraud shading

In this shading algorithm, we calculate the surface normal of each vertex in a polygonal 3D model or average the surface normals of the polygons that meet at each vertex. We apply lighting computations based on a reflection model, for example, the Blinn-Phong reflection model, and then they are performed to produce color intensities at the vertices. For each screen pixel that is covered by the polygonal mesh, color intensities can then be interpolated from the color values calculated at the vertices.

Gouraud shading's strength and weakness lies in its interpolation. If a polygon covers more pixels on the screen space than the number of vertices it has, the interpolating color values from samples of expensive lighting calculations at vertices is less processor-intensive than performing the lighting calculation for each pixel. However, effects such as specular highlights might not be rendered correctly. The following diagram explains Gouraud shading:



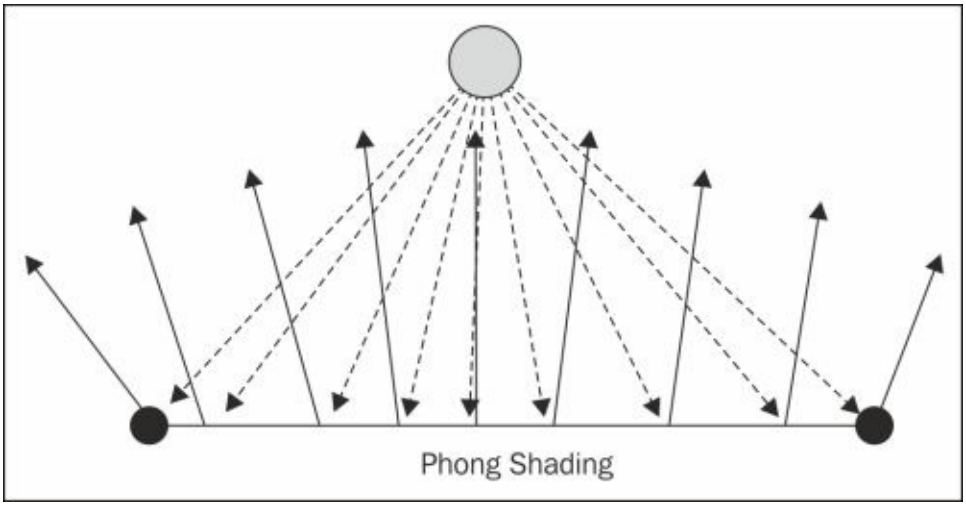
Let's now summarize the steps:

1. Determine the normal at each polygon vertex
2. Apply an illumination model (Lambertian, Blinn-Phong) to each vertex to calculate the vertex intensity
3. Linearly interpolate the vertex intensities over the surface polygon

Phong shading

Phong shading interpolates surface normals across rasterized polygons and computes pixel colors based on the interpolated normals and a reflection model.

The Phong interpolation method works better than Gouraud shading when applied to a reflection model that has small specular highlights such as the Phong reflection model. The following diagram explains Phong shading:

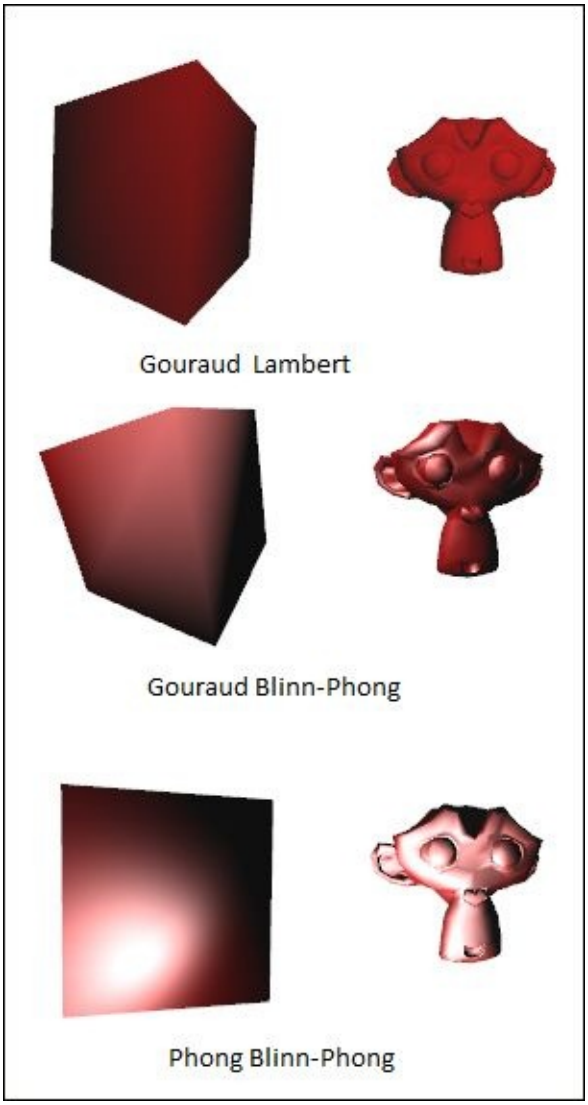


Let's now summarize the steps:

1. Compute a normal, N, for each vertex of the polygon.
2. From the bilinear interpolation, compute a normal, N, for each pixel. (This must be renormalized each time.)
3. From N, compute an intensity, I, for each pixel of the polygon.

Differentiating the shading models

The difference in the output of the shading algorithms is shown in the following diagram. The Gouraud Lambert shaders do not have any specular highlight but Gouraud Blinn-Phong uses specular color. Also, note that the object features are more distinct around the edges in Phong Blinn-Phong shaders:



Implementing Gouraud shading on a Lambertian reflection model

Open the `02-Loading-Model-Gouraud-Lambert.html` file in your favorite text

editor. We have covered only directional lights in our implementation for now and will cover positional lights in Chapter 3, *Loading the Game Scene*.

The shader code for the Gouraud shading for the Lambertian reflection model is as follows:

```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
varying vec3 vColor;

void main(void) {
  gl_FragColor = vec4(vColor, 1.0);
}
</script>

<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;


uniform mat4 mVMatrix;
uniform mat4 pMatrix;
uniform mat4 nMatrix;

uniform vec3 uAmbientColor;
uniform vec3 uLightingDirection;
uniform vec3 uDirectionalColor;
uniform vec3 materialDiffuseColor;
uniform vec3 materialAmbientColor;
varying vec3 vColor;
void main(void) {
  gl_Position = pMatrix  mVMatrix  vec4(aVertexPosition, 1.0);

  vec3 transformedNormal = vec3(nMatrix  vec4(aVertexNormal,1.0));
  vec3 normal=normalize(transformedNormal);
  vec3 unitLightDirection = normalize(uLightingDirection);
  float lambertTerm = max(dot(normal, -unitLightDirection), 0.0);
  vColor = uAmbientColormaterialAmbientColor+ materialDiffuseColor
uDirectionalColor  lambertTerm;

}
</script>
```

In the preceding code, in the vertex shader, we pass the material ambient color, light ambient color, material diffuse color, and light diffuse color in the

`materialAmbientColor`, `uAmbientColor`, `materialDiffuseColor`, and `uDirectionalColor` uniforms respectively. In the first line, we calculate the new transformed vertices and store them in the `gl_position` variable. Then, we calculate `transformedNormal`. The transformed normal is derived by multiplying the normal matrix with the vertex normal. The normal matrix (`nMatrix`) is the transpose of the inverse of the ModelView matrix. Then, we derive `normal`, the unit vector of `transformedNormal`, using the normalize function. From the `normal` vector and uniform light direction, we get the **Lambert** term. Then, the final vertex color is calculated from the ambient and diffuse colors and is passed to the fragment shader.

In the preceding code, we first calculate the transformed normal, and from the transformed normal and light direction, we get the Lambert term:

Lambert term = Vertex normal - Light direction

The control code for the Gouraud shading for the Lambertian reflection model is as follows:

```
function loadModel(url){
  rotateZ=0.0;
  rotateY=0.0;
  $.getJSON(url,function(data){
    vertices = data.vertices;
    var faces=parseJSON(data);
    indices = getIndicesFromFaces(faces);
    if(data.materials.length>0){
      diffuseColor=data.materials[0].colorDiffuse;
      ambientColor=data.materials[0].colorAmbient;
    }
    normals=calculateVertexNormals(vertices,indices);
    initScene();
  });
}
```

The `loadModel(url)` function parses the JSON file, retrieves the vertices data, and gets indices data using the `parseJSON()` function. We also retrieve the ambient and diffuse colors of the model defined in the parse JSON file. We calculate the vertex `normals` array using the `calculateNormals(vertices,indices)` function defined in the utils.js library.

```
function initShaders() {
...
```

```
  shaderProgram.vertexNormalAttribute =
gl.getAttribLocation(shaderProgram, "aVertexNormal");
  gl.enableVertexAttribArray(shaderProgram.vertexNormalAttribute);
...

  shaderProgram.nMatrixUniform =
gl.getUniformLocation(shaderProgram, "nMatrix");
  shaderProgram.ambientColorUniform =
gl.getUniformLocation(shaderProgram, "uAmbientColor");
  shaderProgram.lightingDirectionUniform =
gl.getUniformLocation(shaderProgram, "uLightingDirection");
  shaderProgram.directionalColorUniform =
gl.getUniformLocation(shaderProgram, "uDirectionalColor");
  shaderProgram.materialDiffuseColor =
gl.getUniformLocation(shaderProgram, "materialDiffuseColor");
  shaderProgram.materialAmbientColor =
gl.getUniformLocation(shaderProgram, "materialAmbientColor");
...
}
```

In the preceding code, we activate another vertex buffer, `aVertexNormal`, to store the normal of our vertices. We also get a reference to various uniforms, such as `nMatrix`, to hold the normal matrix and other uniform variables of light direction, material colors, and light colors.

```
function initBuffers() {
...
  vertexNormalBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexNormalBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals),
gl.STATIC_DRAW);
  vertexNormalBuffer.itemSize = 3;
  vertexNormalBuffer.numItems = 24;
...
}
```

In the `initBuffers()` function, we add another buffer to hold the normal data. We calculated this data using vertices and indices of the model.

```
function setMatrixUniforms() {
...
  var invertMatrix = mat4.create();
  mat4.invert(invertMatrix, mvMatrix);
  var normalMatrix = mat4.create();
  mat4.transpose(normalMatrix, invertMatrix)
  gl.uniformMatrix4fv(shaderProgram.nMatrixUniform, false,
normalMatrix);
```

```
normalMatrix);

gl.uniform3f(shaderProgram.materialDiffuseColor,diffuseColor[0],dif
fuseColor[1],diffuseColor[2]);

gl.uniform3f(shaderProgram.materialAmbientColor,ambientColor[0],amb
ientColor[1],ambientColor[2]);
  gl.uniform3f(shaderProgram.ambientColorUniform,1.0,1.0,1.0);
  var lightingDirection = [-0.25,-0.25,-1.0];
  gl.uniform3fv(shaderProgram.lightingDirectionUniform,
lightingDirection);
  gl.uniform3f(shaderProgram.directionalColorUniform,0.2,0.0,0.0);
...
}
```

In the preceding code, we first calculate the normal matrix from `mvMatrix`. A normal matrix is the transpose of the inverse of a matrix. We pass the matrix as a uniform so that we can calculate new transformed normal vectors for illumination calculations. We then pass our different uniform values using the `uniform3f` or `uniform3fv` function. In these functions, `3` stands for the number of elements we want to pass and `f` stands for the data type. The `uniform3f` function is explained in the following code line:

```
void uniform[1234][fi](uint location,x,[y],[z])
```

Here, `[1234]`denotes the number of elements and `[fi]`denotes that we are passing a float or an integer.

Examples are given in the following code lines:

```
gl.uniform3f( shaderProgram.ambientColorUniform, 1.2,1.3,1.4)
gl.uniform4i( shaderProgram.ambientColorUniform, 1,1,1,7)
```

The `uniform3fv` function is explained in the following code line:

```
void uniform[1,234][fi]v(uint location, Array value)
```

Here, `[1,234]` denotes the number of elements in the value parameter and `[fi]` denotes that we are passing a float or an integer.

Examples are given in the following code lines:

```
gl.uniform3fv(shaderProgram.lightingDirectionUniform,
lightingDirection);
```

The `uniformMatrix3fv` function is explained in the following code line:

```
void uniformMatrix[234]fv(uint location, bool transpose, Array)
```

Here, `[234]` denotes 2 x 2 arrays, 3 x 3 arrays, and 4 x 4 arrays. The `transpose` parameter will automatically transpose the matrix while passing its values to the shader, and in the function names (`uniformMatrix3fv`, `uniformMatrix3iv`), the `f` or `i` denotes that we are passing a float or an integer:

```
function drawScene() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  mat4.perspective(pMatrix,40, gl.viewportWidth  gl.viewportHeight,
0.1, 1000.0);
  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix,mvMatrix, [0.0, 0.0,-3.0]);
  /mat4.scale(mvMatrix,mvMatrix,[10,10,10]);
  mat4.rotateY(mvMatrix,mvMatrix,rotateY);
  mat4.rotateX(mvMatrix,mvMatrix,rotateZ);
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
vertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexNormalBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexNormalAttribute,
vertexNormalBuffer.itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.drawElements(gl.TRIANGLES, indices.length,
gl.UNSIGNED_SHORT,0);

}
```

Let's go over the `drawScene` function once again. First, we create a perspective matrix with `40` as the **field of view** (**FOV**), `gl.viewportWidth` or `gl.viewportHeight` as the aspect ratio, `0.1` as the near plane, and `1000.0` as the far plane. We will discuss these parameters in the *Understanding perspective transformations* section in Chapter 5, *Camera and User Interaction*. We initialize the ModelView matrix as the identity matrix. We translate and rotate `mvMatrix` with variable angles controlled by key presses. We associate the shader variable `vertexPositionAttribute` with `vertexPositionBuffer` and `vertexNormalAttribute` with `vertexNormalBuffer`, and then invoke the `drawElements` functions after specifying the `indexBuffer` variable as the active `gl.ELEMENT_ARRAY_BUFFER` constant. The OBJ file has 3*n* indices, *n* being the number of triangles to be drawn; hence, we use `gl.TRIANGLES` as the mode in the

`drawElements` function call.

Implementing Gouraud shading – Blinn-Phong reflection

Open the `02-Loading-Model-Gouraud-Blinn-Phong.html` file in your favorite text editor.

The shader code for the Gouraud shading for the Blinn-Phong reflection model is as follows:

```
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 mVMatrix;
uniform mat4 pMatrix;
uniform mat4 nMatrix;
uniform vec3 uAmbientColor;
uniform vec3 uLightingPosition;
uniform vec3 uDirectionalColor;
uniform vec3 uSpecularColor;
uniform vec3 materialDiffuseColor;
uniform vec3 materialAmbientColor;
uniform vec3 materialSpecularColor;
varying vec3 vColor;
void main(void) {
  vec4 vertexPos4 =  mVMatrix  vec4(aVertexPosition, 1.0);
  vec3 vertexPos = vertexPos4.xyz;
  vec3 eyeVector=normalize(-vertexPos);
  vec3 transformedNormal = vec3(nMatrix  vec4(aVertexNormal,1.0));
  vec3 normal=normalize(transformedNormal);
  vec3 lightDirection = normalize(uLightingPosition);
  float specular = 0.0;
  float lambertTerm = max(dot(normal, -lightDirection), 0.0);
  if(lambertTerm>0.0)
  {
    vec3 halfDir = normalize(-lightDirection + eyeVector);
    float specAngle = max(dot(halfDir, normal), 0.0);
    specular = pow(specAngle, 16.0);
  }
  gl_Position= pMatrix vertexPos4;
  vColor = uAmbientColormaterialAmbientColor+ uDirectionalColor
materialDiffuseColor
lambertTerm+uSpecularColor*materialSpecularColor*specular;

}
</script>
```

The preceding code is the implementation of the Blinn-Phong illumination applied at each vertex normal (Gouraud shading). Hence, the complete calculation is done in the vertex shader. In the control code, we associate `aVertexPosition` and `aVertexNormal` with vertex buffers. In the Blinn-Phong model, we pass light properties, such as ambient, diffuse, specular, and direction, as uniforms from the control code. Material properties such as ambient, diffuse, and specular are also passed as uniforms. The `eyeVector` is the negative of the transformed vertex position. Hence, we first calculate the transformed vertex position and store it in the `vertexPos4` variable. The important thing to note is that the transformation matrix is the 4 x 4 matrix as it stores the translation as well as the rotations. Hence, we convert the vertex position to vector 4. We only need `xyz` and we ignore the w element to get the new vertex position which is then held in `vertexPos`. The `eyeVector` is the negative unit vector of the vertex position. The unit vector is calculated using the `normalize()` function. Hence, the `eyeVector` is equal to `normalize(-vertexPos)`. Then, we calculate the transformed normal unit vector and store it in a variable normal. Then, we calculate the Lambert term using the normal and light direction. If the Lambert term is greater than 0, then we only calculate the Blinn term.

To calculate the Blinn term, we need to first calculate the unit half vector. The **half vector** is the unit vector of the sum of the light vector and eye vector. The angle between the half vector and the vertex normal is calculated by the dot product of the two. Then, the Blinn term is calculated as `specAngle` to the power shininess. Finally, the color is calculated using the following equation:

```
vColor = uAmbientColor * materialAmbientColor+ uDirectionalColor
materialDiffuseColor  lambertTerm+uSpecularColor *
materialSpecularColor * specular;
```

Then, `vColor` is directly passed to the fragment shader.

The control code for the Gouraud shading for the Blinn-Phong reflection model is as follows. In the following code snippet, we retrieve the ambient color from the JSON file:

```
function loadModel(url){
...
  ambientColor=data.materials[0].colorAmbient;
...
}
```

In the following code snippet, we obtain the reference of the light and material ambient colors from the shader code:

```
function initShaders() {
...
  shaderProgram.specularColorUniform =
gl.getUniformLocation(shaderProgram, "uSpecularColor");
...
  shaderProgram.materialSpecularColor =
gl.getUniformLocation(shaderProgram, "materialSpecularColor");
...
}
```

In the following code snippet, we set the value of the material and light specular colors:

```
function setMatrixUniforms() {
...

gl.uniform3f(shaderProgram.materialSpecularColor,specularColor[0],s
pecularColor[1],specularColor[2]);
  gl.uniform3f(shaderProgram.specularColorUniform,1.0,1.0,1.0);
...
}
```

Implementing Phong shading – Blinn-Phong reflection

Open the `02-Loading-Model-Phong-Blinn-Phong.html` file in your favorite text editor. It contains the following code:

```
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
varying vec3 transformedNormal;
varying vec3 vertexPos;

uniform vec3 uAmbientColor;
uniform vec3 uLightingPosition;
uniform vec3 uDirectionalColor;
uniform vec3 uSpecularColor;

uniform vec3 materialDiffuseColor;
uniform vec3 materialAmbientColor;
uniform vec3 materialSpecularColor;

void main(void)  {
  vec3 normal=normalize(transformedNormal);
```

```
  vec3 eyeVector=normalize(-vertexPos);
  vec3 lightDirection = normalize(uLightingPosition);
  float specular = 0.0;

  float directionalLightWeighting = max(dot(normal, -
lightDirection), 0.0);
  if(directionalLightWeighting>0.0)
  {
    vec3 halfDir = normalize(-lightDirection + eyeVector);
    float specAngle = max(dot(halfDir, normal), 0.0);
    specular = pow(specAngle, 4.0);
  }

  vec3 iColor = uAmbientColor*materialAmbientColor+
uDirectionalColor materialDiffuseColor
directionalLightWeighting+uSpecularColor*materialSpecularColor*spec
ular;

  gl_FragColor = vec4(iColor, 1.0);
}
</script>


<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 mVMatrix;
uniform mat4 pMatrix;
uniform mat4 nMatrix;
varying vec3 transformedNormal;
varying vec3 vertexPos;
void main(void) {
  vec4 vertexPos4 =  mVMatrix  vec4(aVertexPosition, 1.0);
  vertexPos = vertexPos4.xyz;
  transformedNormal = vec3(nMatrix  vec4(aVertexNormal,1.0));
  gl_Position= pMatrix *vertexPos4;
}
</script>
```

As discussed earlier, in Phong shading, the normals are calculated after the object is rasterized and calculations are performed for every pixel. Hence, we calculate the vertex and the normal transformations in the vertex shader and then add the transformed values to the fragment shader. Then, the color calculation of the Lambert and Blinn terms are performed in the fragment shader.

Summary

In this chapter, we first discussed simple coloring of objects using vertex colors. In our example, we passed vertex colors and vertex positions to the shader. The shader then interpolated color values of vertices across primitives.

We learned how to export objects from Blender in the OBJ format and render the objects by converting them to the JSON format. The JSON format we used stored indices, colors, normals, and UV maps in the `faces` array.

We also learned that vertex normals are used for illumination models.

We learned about different illumination models such as Lambert and Blinn-Phong. Lambert is used to render objects with diffuse materials and diffuse light. Blinn-Phong is used to render objects with a specular material. These illumination models are used in shading algorithms such as Gouraud and Phong.

In the next chapter, we will render a complete scene of 5000 AD.

Chapter 3. Loading the Game Scene

In the previous chapter, we learned how to load objects exported from Blender and learned about directional lights. We learned many WebGL API calls and now in this chapter, we will put our knowledge to good use. In this chapter, we will not discuss any new WebGL call but we will use our knowledge to create a small game scene. But, before we do that, we will need to perform some code updates.

The code we developed in Chapter 2, *Colors and Shading Languages*, is not capable of handling multiple objects, and so, we will structure that code first.

The following are the topics that we will cover in this chapter:

- Code changes to support multiple objects
- WebGL, a state machine
- Request animation frames
- Load the scene
- Positional lights
- Multiple lights and shaders

Supporting multiple objects

Our starting point will be a new class `Geometry`. The objective of the class is to hold the geometry of our 3D models. We will first understand what type of information associated with each 3D model.

Each 3D model is represented by its `Geometry` class. Each `Geometry` object has information listed as follows:

- **Vertices**: Read from the JSON object.
- **Normals**: Read from the JSON object or calculated from the `vertices` and `indices` arrays.
- **UV map**: Read from the JSON object.
- **Per vertex colors**: Read from the JSON object.
- **Materials**: Read from the JSON object.
- **Indices**: Generated from the faces array.
- **Faces**: The array of the `Face` objects. Each `Face` object represents polygon that forms the geometry.

Note

The classes `Face` and `Geometry` are taken from the three.js library (https://github.com/mrdoob/three.js/blob/master/src/core/), and we have modified them to suit our architecture. We did this since the JSON format we have used in our book is also taken from the three.js library.

Implementing Face.js

Open the `Face.js` file from the `primitive` folder in your editor. We have also added a new function `clone` so that we can create a new `Face` object with its data elements. The constructor defines variables to store information such as vertex index, normal of a polygon, and so on. The `Face.js` file has the following code:

```
Face = function ( a, b, c, normal, color, materialIndex ) {
  this.a = a;// Index to the first vertex of the polygon
  this.b = b;// Index to the second vertex of the polygon
  this.c = c; //Index to the third vertex of the polygon
  this.normal =  normal ;// The index to face normal
  this.vertexNormals = [ ];//Indexes to each vertex normal of the
face
  this.vertexColors = color instanceof Array ?color : [];//Colors
of each vertex
  this.colorIndex = color;
  this.vertexTangents = [];
  this.materialIndex = materialIndex !== undefined ? MaterialIndex
: 0;
};
Face.prototype = {
  constructor: Face,
  clone: function () {
    var face = new Face(this.a, this.b, this.c );
    if(!(this.normal===undefined))
      face.normal=vec3.clone(this.normal );
    face.colorIndex= this.colorIndex;
    face.materialIndex = this.materialIndex;
    var i;
    for ( i = 0; i < this.vertexNormals.length; ++i){
      face.vertexNormals[ i ] = vec3.clone(this.vertexNormals[ i
]);
    }
    for ( i = 0; i<this.vertexColors.length;++i ){
      face.vertexColors[ i ] = this.vertexColors[ i ];
    }
    return face;
  }
};
```

The class variables `a`, `b`, and `c` hold indexes to the `vertices` array defined in the `Geometry` class. The `vertexNormals` array holds indices to the normal array defined in the `Geometry` class for each vertex of that face, as shown in the following code:

```
face.vertexNormals["a"]=3;//3 is the index to the normal array in
the geometry class for the vertex "a" of the face.
```

Implementing Geometry.js

Open the `Geomtery.js` file from the `primitive` folder in your editor. The following code is the constructor of the `Geometry` class. Note the `faces` array; it holds the objects of the faces for that model:

```
Geometry = function () {
  this.vertices = [];
  this.colors = [];
  this.normals = [];
  this.indices=[];
  this.faces = [];
  this.materials=[];

};
```

We have moved the function `getIndicesFromFaces(faces)` from the `parseJSON.js` file to the `Geometry.js` file and renamed it to `indicesFromFaces`, since it populates the class variable `indices`. It now processes the class variable `faces` of the `Geometry` class to update its class variable `indices` as shown in the following code:

```
indicesFromFaces:function (){
  for(var i=0;i<this.faces.length;++i){
    this.indices.push(this.faces[i].a);
    this.indices.push(this.faces[i].b);
    this.indices.push(this.faces[i].c);
  }
},
```

We also moved the `calculateVertexNormals` function from the `utils.js` file to the `geometry.js` file. This function is described in depth in the *Understanding surface normals for lighting calculations* section of Chapter 2, *Colors and Shading Languages*. The only difference in the following code is that it processes class variables `vertices` and `indices` instead of the function parameters:

```
calculateVertexNormals:function(){
  var vertexVectors=[];
  var normalVectors=[];
  var j;
  for(var i=0;i<this.vertices.length;i=i+3){
    var
vector=vec3.fromValues(this.vertices[i],this.vertices[i+1],this.ver
```

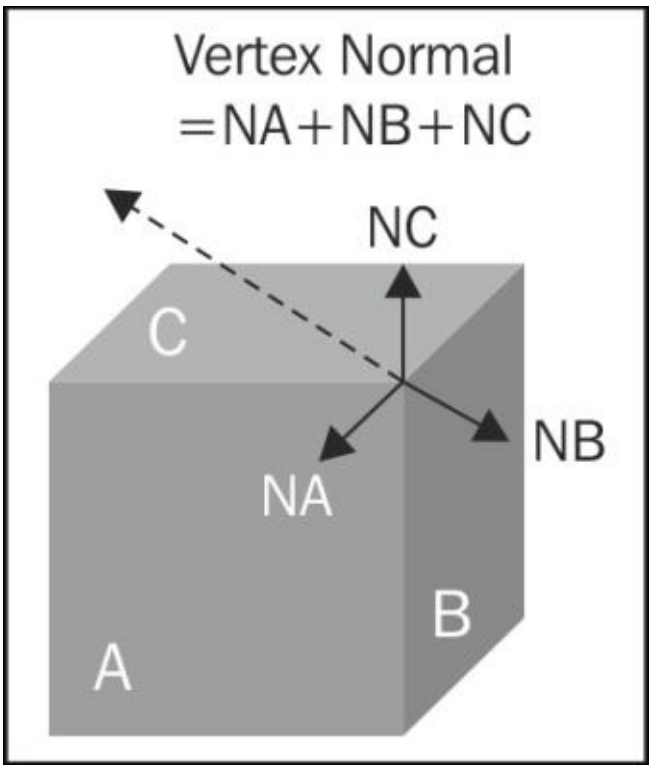
```
tices[i+2]);
    var normal=vec3.create();//Intialiazed normal array
    normalVectors.push(normal);
    vertexVectors.push(vector);
  }
  for(j=0;j<this.indices.length;j=j+3)//Since we are using triads
of indices to represent one primitive
  {
    //v1-v0
    var vector1=vec3.create();

vec3.subtract(vector1,vertexVectors[this.indices[j+1]],vertexVector
s[this.indices[j]]);
    //v2-v1
    var vector2=vec3.create();

vec3.subtract(vector2,vertexVectors[this.indices[j+2]],vertexVector
s[this.indices[j+1]]);
    var normal=vec3.create();
    //cross product of two vector
    vec3.cross(normal, vector1, vector2);
    //Since the normal calculated from three vertices is same for
all the three vertices(same face/surface), the contribution from
each normal to the corresponding vertex  is the same

vec3.add(normalVectors[this.indices[j]],normalVectors[this.indices[
j]],normal);

vec3.add(normalVectors[this.indices[j+1]],normalVectors[this.indice
s[j+1]],normal);

vec3.add(normalVectors[this.indices[j+2]],normalVectors[this.indice
s[j+2]],normal);

  }
  for(j=0;j<normalVectors.length;j=j+1){
  vec3.normalize(normalVectors[j],normalVectors[j]);
  this.normals.push(normalVectors[j][0]);
  this.normals.push(normalVectors[j][1]);
  this.normals.push(normalVectors[j][2]);
  }
},
```

We have added two new functions to the `Geometry` class: `clone` and `morphedVertexNormalsFromObj`. The `clone` function simply copies variables of the `Geometry` class to create a new `geometry` object.

The other function `morphedVertexNormalsFromObj` is important. The exported `obj` file from Blender contains information on normals of every vertex of every face of the model. In such cases, we do not need to calculate the normal. This information is also encoded in our JSON file. While we parse our JSON file, we populate each face with its vertex normals. The following code snippet is taken from the `parseJSON.js` file to confirm the preceding statement:

```
if ( hasFaceVertexNormal ) {
  var aVertices=["a","b","c","d"]
  for ( i = 0; i < nVertices; i++ ) {
    var aVertex=aVertices[i];
    normalIndex = faces[ offset ++ ] * 3;
    normal = vec3.fromValues(normals[ normalIndex ++ ],normals[
normalIndex ++ ],normals[ normalIndex ]);
    face.vertexNormals[aVertex]= normal;
  }
}
```

The face contains normal data per vertex. The final normal of the shared vertex has contribution from each face. Hence, the normal of the vertex a is the vector sum of **NA**, **NB**, and **NC**, as shown in the following diagram:

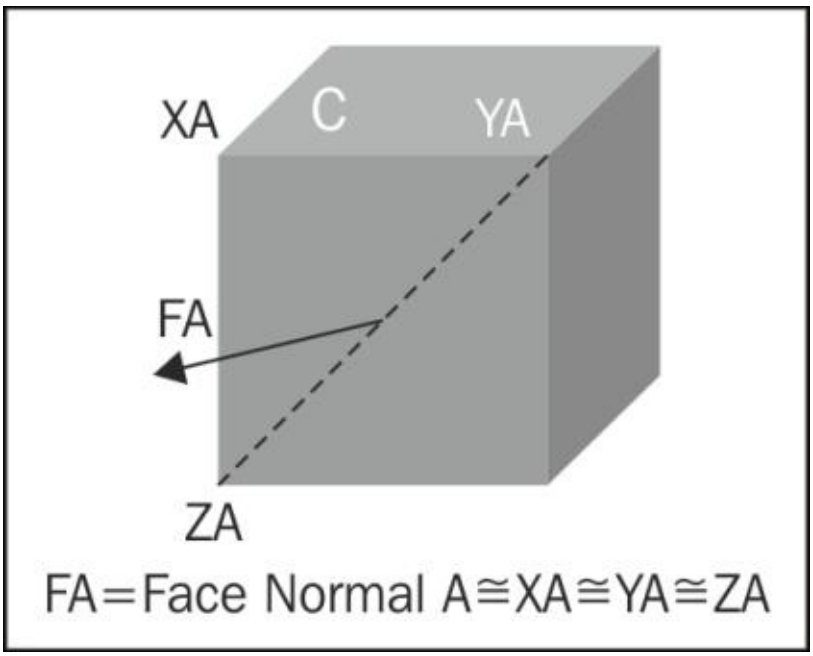


The function `morphedVertexNormalsFromObj` calculates the final normal of each vertex, as shown in the following code:

```
morphedVertexNormalsFromObj:function(){
  var vertexVectors=[];
  var normalVectors=[];

  for(var i=0;i<this.faces.length;i=i+1){
    //This condition checks if face normal is defined. Then, we
first clone the face normal for each vertex. The OBJ file gives
vertex normals.
    if(!
(this.faces[i].normal==undefined)&&this.faces[i].vertexNormals.leng
th==0){

this.faces[i].vertexNormals["a"]=vec3.clone(this.faces[i].normal);

this.faces[i].vertexNormals["b"]=vec3.clone(this.faces[i].normal);

this.faces[i].vertexNormals["c"]=vec3.clone(this.faces[i].normal);
    }
    if(normalVectors[this.faces[i].a]===undefined)
    {

normalVectors[this.faces[i].a]=vec3.clone(this.faces[i].vertexNorma
ls["a"]);
    }
    else{

vec3.add(normalVectors[this.faces[i].a],normalVectors[this.faces[i]
.a],this.faces[i].vertexNormals["a"])
    }

    //Repeat the above code for b and c vertex as well.

  }
  this.normals=[];
  for(var j=0;j<normalVectors.length;j=j+1){
    vec3.normalize(normalVectors[j],normalVectors[j]);
    this.normals.push(normalVectors[j][0]);
    this.normals.push(normalVectors[j][1]);
    this.normals.push(normalVectors[j][2]);
  }
},
```

In the preceding code, we iterated over the `faces` array. If a face normal is

defined and vertex normals are not defined for that face, we copy the face normal on each vertex normal (`a`, `b`, `c`). In the following diagram, the vertex normals will be the same as the face normal:



Elements of the normal array have one-to-one correspondence to the `vertices` array. Hence, if a normal is already defined for the vertex index `this.faces[i].a`, then add the new normal to the existing normal, or simply copy the normal at that vertex index. This can be done using the following code:

```
if(normalVectors[this.faces[i].a]===undefined)
{

normalVectors[this.faces[i].a]=vec3.clone(this.faces[i].vertexNorma
ls["a"]);
}
else{

vec3.add(normalVectors[this.faces[i].a],normalVectors[this.faces[i]
.a],this.faces[i].vertexNormals["a"])
}
```

In the last loop, we iterate over the newly created `normalVectors` array. First, we normalize the vector and then copy each element to the `normals` array of the `Geometry` class.

Implementing parseJSON.js

We have only done a few changes in the `parseJSON.js` file present in the `primitive` folder. Open it in your favorite editor. Earlier, this class returned the initialized `faces` array from the parsed JSON, but now, it initializes the `Geometry` object that in turn holds the `faces` array, as shown in the following code:

```
var geometry=new Geometry();
```

Now, instead of initializing individual data elements, we initialize variables of the `Geometry` class. Also, before returning the `Geometry` object, we initialize its data members by invoking the functions that we explained earlier. This is shown in the following code:

```
geometry.vertices=vertices;
geometry.materials=data.materials;
geometry.indicesFromFaces();
//Normal information is present in JSON object then invoke
morphedVertexNormalsFromObj else invoke calculateVertexNormals.
if(data.normals.length>0){
  geometry.morphedVertexNormalsFromObj();
}else{
  geometry.calculateVertexNormals();
}
```

Implementing StageObject.js

The new class `StageObject` that we have added basically holds information to render geometry on the stage or on our scene. We have used the word "Stage" for our game space. The purpose of this class is to initialize a `Geometry` object and initialize buffer objects for that `Geometry` class. Also, this class contains the location and rotation of the object with respect to our scene. The following code is the constructor of the `StageObject` class:

```
StageObject=function(){
  this.name="";
  this.geometry=new Geometry();
  this.location=vec3.fromValues(0,0,0);
  this.rotationX=0.0;
  this.rotationY=0.0;
  this.rotationZ=0.0;
  this.ibo=null;//Index buffer object
  this.vbo=null;//Buffer object for vertices
  this.nbo=null;//Buffer Object for normals
  this.diffuseColor=[1.0,1.0,1.0,1.0];
  this.ambientColor=[1.0, 1.0, 1.0];
  this.specularColor=[0.000000000000001, 0.0000000000000001,
0.0000000000000001];
};
```

Notice the class variables `ibo`, `nbo`, and `vbo`. These variables will hold reference to the index buffer object and vertex buffer objects for that geometry.

The `loadObject` function of the class invokes the `parseJSON` object and initializes color values for the geometry. This function also initializes the name of the geometry from the metadata information from the parsed JSON object. If you look into the code from Chapter 2, *Colors and Shading Languages*, this code has been moved from the HTML file to this class:

```
loadObject: function (data){
  this.geometry=parseJSON(data);
  this.name=data.metadata.sourceFile.split(".")[0];
  if(this.geometry.materials.length>0){
    if(!(this.geometry.materials[0].colorDiffuse===undefined))
    this.diffuseColor=this.geometry.materials[0].colorDiffuse;
    if(!(this.geometry.materials[0].colorAmbient===undefined))
    this.ambientColor=this.geometry.materials[0].colorAmbient;
    if(!(this.geometry.materials[0].colorSpecular===undefined))
    this.specularColor=this.geometry.materials[0].colorSpecular;
```

```
  }
},
```

The `createBuffers` function initializes buffer objects for this geometry. Earlier, we had an `initBuffers` function that initialized buffer objects. But now, since we will have to load multiple models from multiple JSON files, we have moved this code to initialize buffers for each geometry:

```
createBuffers:function(gl){
  this.vbo = gl.createBuffer();
  this.ibo = gl.createBuffer();
  this.nbo = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.ibo);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
Uint16Array(this.geometry.indices), gl.STATIC_DRAW);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
  gl.bindBuffer(gl.ARRAY_BUFFER, this.vbo);
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(this.geometry.vertices), gl.STATIC_DRAW);
  this.vbo.itemSize = 3;
  this.vbo.numItems = this.geometry.vertices.length/3;
  gl.bindBuffer(gl.ARRAY_BUFFER, this.nbo);
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(this.geometry.normals), gl.STATIC_DRAW);
  this.nbo.itemSize = 3;
  this.nbo.numItems = this.geometry.normals.length/3;
},
```

The last function of this class is `clone`. This function creates copy of its local variables and returns a new `StageObject`. The `clone` function invokes the `clone` function of the `Geometry` class and the later invokes the `clone` function of the `Face` class, as shown in the following code:

```
clone:function(){
  var stageObject=new StageObject();
  stageObject.geometry=this.geometry.clone();

  var i;
  for(i=0;i<this.diffuseColor.length;++i){
    stageObject.diffuseColor[i]=this.diffuseColor[i];
  }
  for(i=0;i<this.ambientColor.length;++i){
    stageObject.ambientColor[i]=this.ambientColor[i];
  }
  for(i=0;i<this.specularColor.length;++i){
```

```
    stageObject.specularColor[i]=this.specularColor[i];
  }
  stageObject.rotationX=this.rotationX;
  stageObject.rotationY=this.rotationY;
  stageObject.rotationZ=this.rotationZ;
  stageObject.loaded=true;
  return stageObject;

}
```

Implementing Stage.js

The `Stage` class holds the array of stage objects as well as the initialized WebGL context in the variable `gl`, as shown in the following code:

```
Stage = function (gl) {
    this.stageObjects=[];
    this.gl=gl;
};
```

The `addModel` function does two things; it pushes the `stageObject` to the array and also initializes the buffers by invoking the `createBuffers` function of the `stageObject`:

```
addModel:function(stageObject){

  if(!(this.gl===undefined))  {
    stageObject.createBuffers(this.gl);

  }
  this.stageObjects.push(stageObject);
},
```

The following are a few changes that we performed in the preceding code:

- The `parseJSON` function now returns the `Geometry` object instead of the `faces` array.
- The `geometry` object calculates normal data from the `vertexNormals` array stored in the `faces` array. The `StageObject` class holds the reference of the `Geometry` object and is responsible for creating buffers for the object.
- The `Stage` class holds references to all the `stageObjects`.

Using the architectural updates

Open the `03-Loading-Object-Architectural-Updates.html` file in your favorite editor. Let's now look at the changes in our main code to use the previous changes. The end result is not any different than the output of Chapter 2, *Colors and Shading Languages*. But, the code changes will now enable us to load multiple models that we will see in the next section.

In this section, we have just reconstructed our code to handle multiple JSON objects. Hence, there are no changes in the shader script.

Understanding the main code

The functions `initGL`, `initShaders`, and `getShaders` have no changes. The `start` function initializes the `Stage` object and invokes the `addStageObject` function to load our JSON model, as shown in the following code:

```
function start() {
  var canvas = document.getElementById("squareWithDrawArrays");
  initGL(canvas);
  initShaders();
  stage=new Stage(gl);
  addStageObject("model/Monkey.json",[0.0,0.0,0.0],0.00,0.0,0.0);
  initScene();
}
```

The `addStageObject` function has three parameters: `url`, `location`, and the rotation of the object. After the JSON file is loaded, we initialize the `stageObject`, add it to the main `stage` object, and invoke the `drawScene` function, as shown in the following code:

```
function addStageObject(url,location,rotationX,rotationY,rotationZ)
{
  $.getJSON(url,function(data){
    var stageObject=new StageObject();
    stageObject.loadObject(data);
    stageObject.location=location;
    stageObject.rotationX=rotationX;
    stageObject.rotationY=rotationY;
    stageObject.rotationZ=rotationZ;
    stage.addModel(stageObject);
    //Invoke drawScene once the object is loaded.
    drawScene();
  });
```

```
}
```

The following code is to initialize general scene uniforms that have been split in two functions; one for light uniforms and the second for perspective, ModelView matrices:

```
function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false,
pMatrix);
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false,
mvMatrix);
  var invertMatrix = mat4.create();
  mat4.invert(invertMatrix, mvMatrix);
  var normalMatrix=mat4.create();
  mat4.transpose(normalMatrix, invertMatrix)
  gl.uniformMatrix4fv(shaderProgram.nMatrixUniform, false,
normalMatrix);
}
function setLightUniform(){
  gl.uniform3f(shaderProgram.ambientColorUniform,0.1,0.1,0.1);
  gl.uniform3f(shaderProgram.specularColorUniform,1.0,1.0,1.0);
  var lightingDirection = [0,-1.25,-1.25];
  gl.uniform3fv(shaderProgram.lightingDirectionUniform,
lightingDirection);
  gl.uniform3f(shaderProgram.directionalColorUniform,0.5,0.5,0.5);
}
```

Finally, we are left with our `drawScene` function. Since we are rendering a single object, we only referenced the first object of the `stageObjects` array. We activated its buffers using the `vertexAttribPointer` API call for vertices and normals. Then, we initialized the material's color (diffuse, specular, and ambient) uniforms. Now, instead of using the global JavaScript variables `vertexBuffer` and `indexBuffer`, we use the `stageObject` array's buffer objects such as `stageObject.ibo`, `stageObject.vbo`, and `stageObject.nbo`. The following is the code for the `drawScene` function:

```
function drawScene() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  mat4.perspective(pMatrix,degToRadian(55), gl.viewportWidth /
gl.viewportHeight, 0.1, 1000.0);
  setLightUniform();
  var i=0;
  mat4.translate(mvMatrix,mvMatrix,
stage.stageObjects[i].location);
```

```
  mat4.rotateX(mvMatrix,mvMatrix,stage.stageObjects[i].rotationX);
  mat4.rotateY(mvMatrix,mvMatrix,stage.stageObjects[i].rotationY);
  mat4.rotateZ(mvMatrix,mvMatrix,stage.stageObjects[i].rotationZ);
  setMatrixUniforms();
  gl.bindBuffer(gl.ARRAY_BUFFER, stage.stageObjects[i].vbo);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
stage.stageObjects[i].vbo.itemSize, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ARRAY_BUFFER,stage.stageObjects[i].nbo);
  gl.vertexAttribPointer(shaderProgram.vertexNormalAttribute,
stage.stageObjects[i].nbo.itemSize, gl.FLOAT, false, 0, 0);

gl.uniform3f(shaderProgram.materialDiffuseColor,stage.stageObjects[
i].diffuseColor[0],stage.stageObjects[i].diffuseColor[1],stage.stag
eObjects[i].diffuseColor[2]);

gl.uniform3f(shaderProgram.materialAmbientColor,stage.stageObjects[
i].ambientColor[0],stage.stageObjects[i].ambientColor[1],stage.stag
eObjects[i].ambientColor[2]);

gl.uniform3f(shaderProgram.materialSpecularColor,stage.stageObjects
[i].specularColor[0],stage.stageObjects[i].specularColor[1],stage.s
tageObjects[i].specularColor[2]);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
stage.stageObjects[i].ibo);
  gl.drawElements(gl.TRIANGLES,
stage.stageObjects[i].geometry.indices.length,
gl.UNSIGNED_SHORT,0);
}
```

Understanding WebGL – a state machine

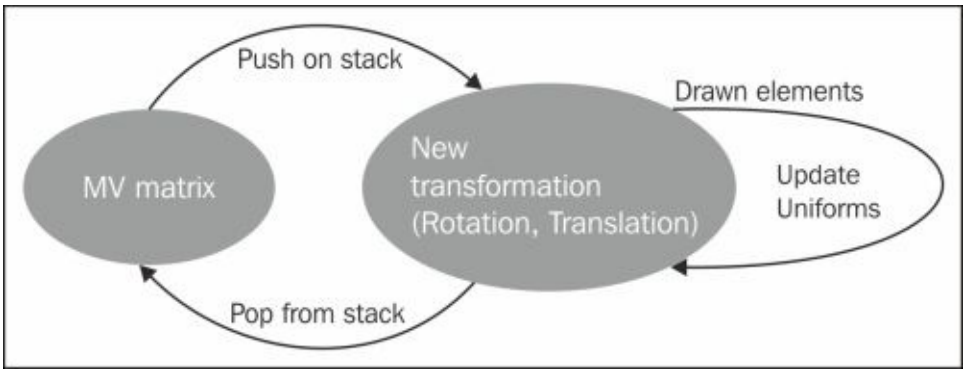
WebGL does not render or work on multiple buffers at a single point in time. When we load multiple objects, we will have multiple buffers. To render all those buffers, we will have to activate each one of them one by one. WebGL is like a state machine; you can always retrieve the active buffer information and the state of the rendering pipeline using functions such as `getParameter`, `getBufferParameter`, and `isBuffer` as shown in the following code:

```
gl.getParameter(gl.ARRAY_BUFFER_BINDING)// It retrieves a reference
to the currently-bound VBO
gl.getParameter(gl.ELEMENT_ARRAY_BUFFER_BINDING)// It retrieves a
reference to the currently bound IBO
gl.getBufferParameter(gl.ARRAY_BUFFER,gl.BUFFER_SIZE);//Get is the
size of the requested buffer
gl.getBufferParameter(gl.ARRAY_BUFFER,gl.BUFFER_STATUS);//Get is
the status of the requested buffer
```

The functions can be used for vertex and index buffer objects. Hence, the preceding functions can be used with `gl.ELEMENT_ARRAY_BUFFER` as a parameter.

Using mvMatrix states

Since we will have multiple objects in the scene, each object will have their own local transformations. Hence, we will not like to lose our initial mvMatrix's state. Our initial `mvMatrix` has our viewer's transformation, which will be applied to each object and then each of their local transformation will be applied. We store our object's local transformation in the `StageObject` class' variables' location with the *x*, *y*, and *z* values as `rotationX`, `rotationY`, and `rotationZ` respectively. Before applying these transformations, we push our matrix to a stack and pop it once the rendering of the object has finished, as shown in the following diagram:



In subsequent chapters, when we will be dealing with complex geometries with multiple objects, we will use matrix stacks to store the parent object's transformations before applying transformations to child objects. This way, the child objects will be rendered with respect to their parent object. The following code implements the matrix stack:

```
var matrixStack = [];
function pushMatrix() {
  var copy = mat4.create();
    mat4.copy(copy,mvMatrix);
  matrixStack.push(copy);
}
function popMatrix() {
  if (matrixStack.length == 0) {
    throw"Invalid popMatrix!";
  }
  mvMatrix = matrixStack.pop();
}
```

Also, let's look at how we use this stack in our `drawScene` function:

```
function drawScene() {
.........................
  for(var i=0;i<stage.stageObjects.length;++i){
.......................
    pushMatrix();
    mat4.translate(mvMatrix,mvMatrix,
stage.stageObjects[i].location);

mat4.rotateX(mvMatrix,mvMatrix,stage.stageObjects[i].rotationX);

mat4.rotateY(mvMatrix,mvMatrix,stage.stageObjects[i].rotationY);

mat4.rotateZ(mvMatrix,mvMatrix,stage.stageObjects[i].rotationZ);
....//Rendering code(Buffer Activation and
drawElementscall..............
    popMatrix();
  }
}
```

Understanding request animation frames

Most HTML5-enabled browsers have implemented a new function `window.requestAnimationFrame()`. This function is a better alternative to the `setTimeout` and `setInterval` functions. The function is designed to invoke rendering functions in your gaming application, such as the `drawScene` function in our case. It executes in a safe way. It executes the target function only when the browser/tab has focus, which saves precious GPU resources. Using this function, we can obtain a rendering cycle that goes as fast as the browsers allow.

To use this function, we included a new file in our code, `webgl-utils.js` (https://code.google.com/p/webglsamples/source/browse/book/extension/webgl-utils.js). This small library has many useful functions but we will use only one function from it. We like this library since it has implemented this function in a cross-browser fashion. If a browser does not support it, then it invokes the `setTimeout` function. The following is the code snippet from the `webgl-utis.js` library:

```
window.requestAnimFrame = (function() {
  return window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame ||
    window.oRequestAnimationFrame ||
    window.msRequestAnimationFrame ||
    function(/* function FrameRequestCallback  callback,
DOMElement Element */ element) {
    window.setTimeout(callback, 1000/60);
  };
})();
```

The following code demonstrates its implementation in our application. The `initScene` function invokes the `tick` function that later invokes itself by the `requestAnimFrame` function at a fixed frame rate:

```
function initScene(){

  tick();
}
function tick(){
  requestAnimFrame(tick);
  drawScene();
```

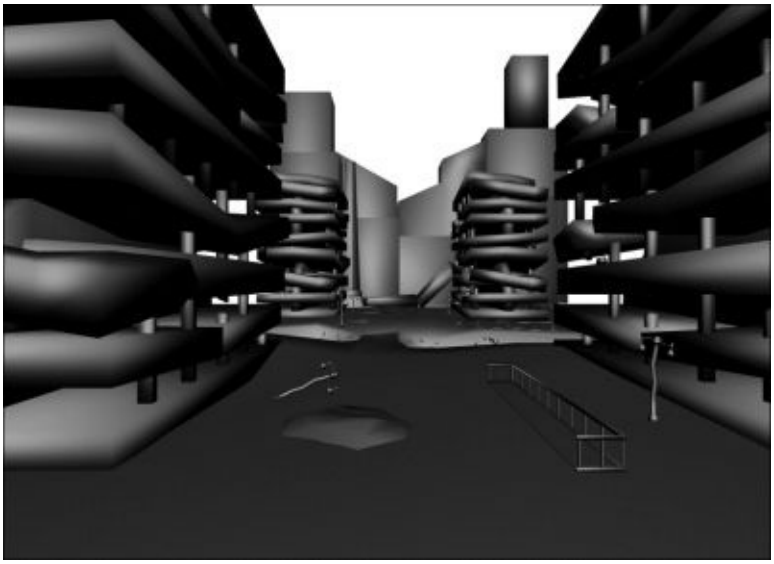
```
  drawScene();
}
```

Note

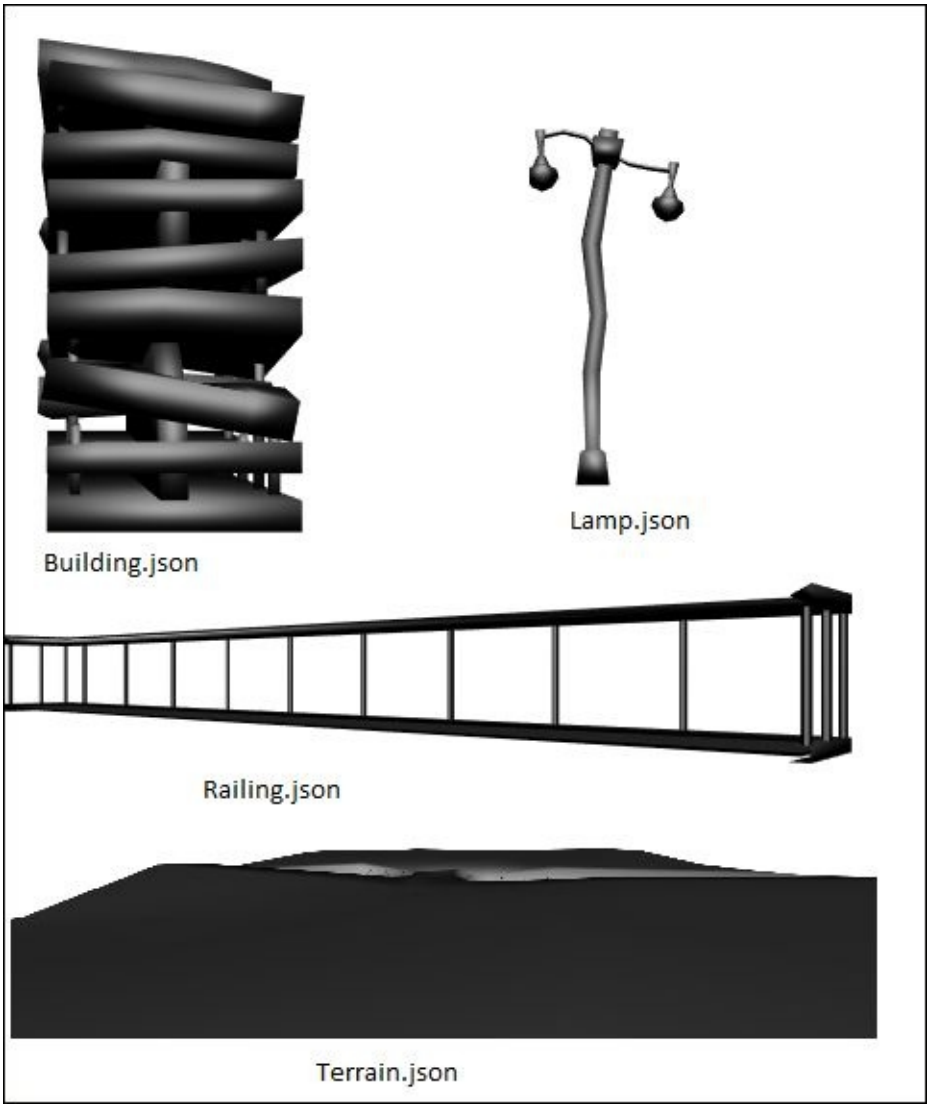
The `tick` function will be extended in Chapter 5, *Camera and User Interaction*, to calculate the elapsed time for animation and timing strategies.

Loading the scene

Our objective is to create a scene of 5000 AD similar to the following screenshot:



We will use these objects to draw the scene. These objects were created using Blender and exported as shown in Chapter 2, *Colors and Shading Languages*.



Open the `03-Loading-Scene.html` file in your favorite editor. If you observe the scene, you will see that there are four buildings, six street lamps, and two railings. Hence, we added the `cloneObjects` function that clones the objects and adds them to the scene. The `cloneObjects` function invokes the function `clone` of the SceneObject's class, which invokes the `clone` function of the `Geometry` class. The following is the code snippet from the `03-Loading-Scene.html` file:

```
function start() {
.....................................
  addStageObject("model/obj/terrain.json",
[0.0,0.0,0.0],0.00,0.0,0.0);
  addStageObject("model/obj/building.json",
```

```
[55.0,0.0,170],0.0,0.0,0.0);
  addStageObject("model/obj/giantPillar.json",
[-60.0,0.0,-200],0.0,0.0,0.0);
  addStageObject("model/obj/pillarBroken.json",
[10.0,0.0,-200.00],0.0,0.0,0.0);
  addStageObject("model/obj/streetlamp.json",
[20.0,0.0,200.00],0.0,0.0,0.0);
  addStageObject("model/obj/cityLine.json",
[12.0,0.0,-335.0],0,3.14,0.0);
  addStageObject("model/obj/railing.json",
[0.0,5.0,200.0],0.0,0.0,0.0);
  addStageObject("model/obj/dump.json",
[-40.0,0.0,-30.0],0.0,0.0,0.0);
  initScene();
}
function addStageObject(url,location,rotationX,rotationY,rotationZ)
{
  $.getJSON(url,function(data){
    ........................................
    cloneObjects(stageObject)
    ........................
  });
}
function cloneObjects(stageObject){){
  if(stageObject.name=="building"){
    var building1=stageObject.clone();
    ................
    building1.location=[-85.0,0.0,170.0];
.........
    stage.addModel(building1);
    .....
  }
  if(stageObject.name=="streetlamp"){
.........
    var streetlamp1=stageObject.clone();
    ............
    streetlamp1.location=[20.0,0.0,70.0];
    streetlamp1.rotationX=degToRadian(-90);
    ............
    stage.addModel(streetlamp1);
    ............
  }
  if(stageObject.name=="railing"){

    var railing1=stageObject.clone();
    var railing2=stageObject.clone();
    railing1.location=[40.0,2.0,0.0];
    railing1.rotationY=degToRadian(-120);
```

```
    railing2.location=[-30.0,5.0,-100.0];
    railing2.rotationY=degToRadian(-65);
    stage.addModel(railing1);
    stage.addModel(railing2);
  }
  if(stageObject.name=="dump"){
    ........
  }
}
```

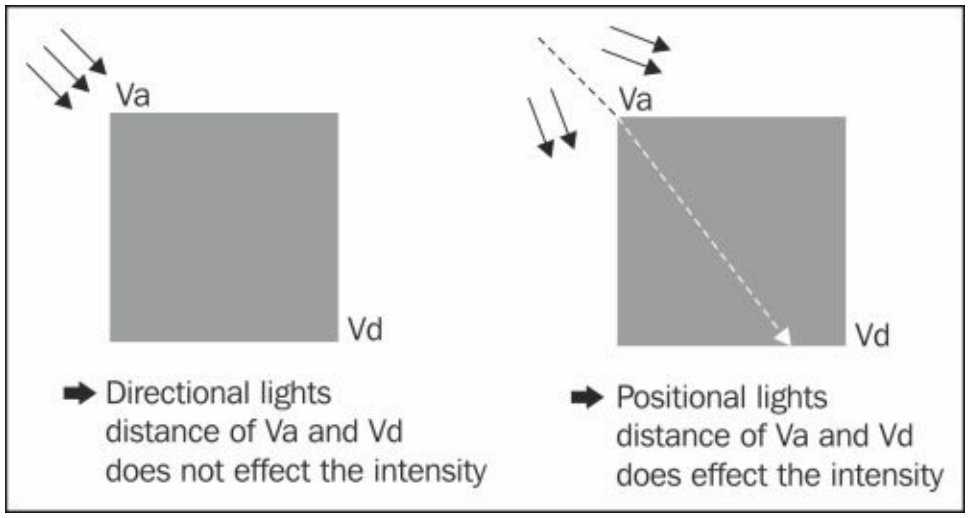
Also, we do not wait for all objects to load before invoking the `drawScene` function. It is invoked at a regular frame rate from the `tick` function. The `drawScene` function also stores the state of the `mvMatrix` variable on stack before applying transforms for each object.

Understanding positional lights

In the scene that we developed in the preceding section, we would like two of the lamps to light up. There is already one global light in the scene but we want two positional lights now.

The directional lights are parallel since they are assumed to be at an infinite distance and are generally the global light in a scene. The intensity of a directional light on a vertex is not dependent on its distance but on its orientation. Hence, only the normal direction is used to calculate its intensity. For positional lights, both the distance and orientation are involved in the intensity calculations.

The difference between directional and positional lights is that the directional lights are defined by a direction vector, whereas positional lights are defined by their locations in *x*, *y*, and *z* coordinates. The difference between directional and positional lights is shown in the following diagram:



For positional lights, we calculate the light direction for a particular vertex and the result is stored in the varying, `uLightRayLamp1`. This variable is then used by the fragment shader. The rest of the calculation remains the same as with the directional light. Hence, for positional lights, we calculate the light direction for each vertex. We call this a ray.

We pass the position of light as a uniform to the vertex shader. Then, the vector is calculated per vertex and is passed to the fragment shader as a varying variable.

The following code explains the `uLightPositionLamp1` uniform that is passed to the vertex shader, and using the variable we calculate the `uLightRayLamp1` varying for use in the fragment shader:

```
vec4 newLightPosition1=mVMatrix * vec4(uLightPositionLamp1, 1.0);
uLightRayLamp1=vertexPos-newLightPosition1.xyz;
```

Lighting up the scene with lamps

Open the `03-Loading-Scene-With-Lamps.html` file in your favorite editor. The following sections in the chapter will explore the code changes to add positional lights to our stage.

The vertex shader

The following code shows the steps to be performed to calculate the light ray on a vertex for two positional lights (street lamps in our scene). We added two uniforms for light positions (`uLightPositionLamp1` and `uLightPositionLamp2`). First, the position (`uLightPositionLamp1`) of the light is transformed by multiplying it by the ModelView matrix (`newLightPosition1`). Then, the ray is calculated by subtracting the transformed light position (`newLightPosition1`) from the transformed vertex position (`vertexPos`). The light ray is stored in a varying variable (`uLightRayLamp1`) to be passed to the fragment shader:

```
<script id="shader-vs" type="x-shader/x-vertex">
.............................
  varying vec3 vertexPos;
  uniform vec3 uLightPositionLamp1;
  uniform vec3 uLightPositionLamp2;
.........................
  varying vec3 uLightRayLamp1;
  varying vec3 uLightRayLamp2;
  void main(void) {
    vec4 vertexPos4 =  mVMatrix * vec4(aVertexPosition, 1.0);
    vertexPos = vertexPos4.xyz;
    vec4 newLightPosition1=mVMatrix  vec4(uLightPositionLamp1,
1.0);
    vec4 newLightPosition2=mVMatrix  vec4(uLightPositionLamp2,
1.0);
    uLightRayLamp1=vertexPos-newLightPosition1.xyz;
    uLightRayLamp2=vertexPos-newLightPosition2.xyz;
    transformedNormal = vec3(nMatrix  vec4(aVertexNormal,1.0));
    gl_Position= pMatrix vertexPos4;
  }
</script>
```

The rest of the calculation in the fragment shader is the same as that for directional light, where the light ray represents the direction.

The fragment shader

In the following code, the `uPositionalColor1` and `uPositionalColor2` uniforms are the diffuse colors of the positional light. The `uLightRayLamp1` and `uLightRayLamp2` uniforms are the varying passed from the vertex shader. The `directionalLightWeighting1` float data type is a Lambert term calculated from the normalized light ray vector. The final diffuse light intensity (`iDiffuse`) is calculated after adding the diffuse intensity of the positional light to the diffuse intensity from other lights. This is shown in the following code:

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec3 transformedNormal;
  varying vec3 vertexPos;
  uniform vec3 uAmbientColor;
  uniform vec3 uLightingDirection;
  uniform vec3 uDirectionalColor;
  uniform vec3 uSpecularColor;
  uniform vec3 uPositionalColor1;
  uniform vec3 uPositionalColor2;
  uniform vec3 materialDiffuseColor;
  uniform vec3 materialAmbientColor;
  uniform vec3 materialSpecularColor;
  varying vec3 uLightRayLamp1;
  varying vec3 uLightRayLamp2;
  void main(void)  {
    vec3 normal=normalize(transformedNormal);
    vec3 eyeVector=normalize(-vertexPos);
    vec3 lightDirection = normalize(uLightingDirection);
    vec3 lightDirection1 = normalize(uLightRayLamp1);
    vec3 lightDirection2 = normalize(uLightRayLamp2);
    vec3 iAmbient=uAmbientColor*materialAmbientColor;
    vec3 iDiffuse=vec3(0.0,0.0,0.0);
    vec3 iSpecular=vec3(0.0,0.0,0.0);
    float specular = 0.0;
    float directionalLightWeighting = max(dot(normal, -
lightDirection), 0.0);
    iDiffuse+=uDirectionalColor materialDiffuseColor
directionalLightWeighting;
    float directionalLightWeighting1 = max(dot(normal, -
lightDirection1), 0.0);
    iDiffuse+=uPositionalColor1 materialDiffuseColor
directionalLightWeighting1;
    float directionalLightWeighting2 = max(dot(normal, -
lightDirection2), 0.0);
    iDiffuse+=uPositionalColor2 materialDiffuseColor
directionalLightWeighting2;
    if(directionalLightWeighting>0.0)
```

```
    {
      vec3 halfDir = normalize(-lightDirection + eyeVector);
      float specAngle = max(dot(halfDir, normal), 0.0);
      specular = pow(specAngle, 4.0);
      iSpecular+=uSpecularColor*materialSpecularColor*specular;
    }
    vec3 iColor =iAmbient+ iDiffuse+iSpecular;
    gl_FragColor = vec4(iColor, 1.0);
  }
</script>
```

Understanding the main code

We changed our `initShaders` function to get the reference of the new uniform variables. We then added this function to our shaders (`diffuseColor` and position of lights):

```
function initShaders() {
.....................
  shaderProgram.uLightPositionLamp1 =
gl.getUniformLocation(shaderProgram, "uLightPositionLamp1");
  shaderProgram.uPositionalColor1 =
gl.getUniformLocation(shaderProgram, "uPositionalColor1");
  shaderProgram.uLightPositionLamp2 =
gl.getUniformLocation(shaderProgram, "uLightPositionLamp2");
  shaderProgram.uPositionalColor2 =
gl.getUniformLocation(shaderProgram, "uPositionalColor2");
.............................
}
Function setLightUniform sets the values of position and diffuse
color of the lights that we have added to our shaders.
function setLightUniform(){
.................
  var lightingPosition = [20.00,5.00,200.00];
  gl.uniform3fv(shaderProgram.uLightPositionLamp1,
lightingPosition);

gl.uniform3f(shaderProgram.uPositionalColor1,lightColor[0],lightCol
or[1],lightColor[2]);
  var lightingPosition1 = [20.00,5.00,-100.0];
  gl.uniform3fv(shaderProgram.uLightPositionLamp2,
lightingPosition1);
  gl.uniform3f(shaderProgram.uPositionalColor2,1.0,1.0,1.0);
}
```

To show the use of animation in our scene, we have added a little effect by

blinking the first light. This will also clearly show you which objects are affected by the positional light. If you notice in the `setLightUniform` function, we did not give a constant value to our `uPositionalColor` uniform but we added a variable `lightColor`. The value of the color changes after every two seconds. The `totalElapsedTime` variable is calculated in the `animate` function. The `animate` function is invoked from the `tick` function, as shown in the following code:

```
var lastTime = 0;
var totalElapsedTime=0;
function animate() {
  var timeNow = new Date().getTime();
  if (lastTime != 0) {
    totalElapsedTime+= (timeNow – lastTime);
    if(totalElapsedTime>=2000){
      if(lightColor[0]==0.5){
        lightColor=[0.0,0.0,0.0];
      }
      else{
        lightColor=[0.5,0.5,0.5];
      }
      totalElapsedTime=0;
    }

  }
  lastTime = timeNow;
}
function tick(){
  requestAnimFrame(tick);
  drawScene();
  animate();
}
```

Multiple lights and shaders

The provided solution in the previous section to add lights to our scene is not a scalable solution because if we wanted to add four or more lights, we would have had to add more uniforms to our shader. WebGL limits the amount of storage for uniforms. If we exceed the limit, we would get a compile-time or a runtime error. You can query the number of allowed uniforms using the following functions:

```
gl.getParameter(gl.MAX_VERTEX_UNIFORM_VECTORS);
gl.getParameter(gl. MAX_FRAGMENT_UNIFORM_VECTORS);
```

Also, the ambient color from all lights need not be calculated in the shaders as the ambient component is independent of distance and orientation of the object. We can sum the ambient color from all lights and pass the sum to the shader. Even if we use a single ambient color uniform in our shader for all lights, we would need two to three uniforms (position/direction, `diffuseColor`, `specularColor`) per light.

Adding multiple lamps

Open the `03-Loading-Scene-With-Lamps-Arrays.html` file in your favorite editor. The solution to the problem is adding arrays to our shaders. There are two important things to note about the use of arrays in the shading language. The first is that many WebGL implementations will not allow an array to be indexed with a variable or with an unknown value at compile time. That is, WebGL only mandates that array indexing is to be supported by constant integral expressions.

The other note about arrays is that there is no syntax in the shading language to initialize an array at the creation time. The elements of the array need to be initialized one by one, and also, arrays cannot be qualified as `const`. So, even if you declare an array as `int a[];`, you need to specify the size as `int a[5];` before indexing it.

When we discuss arrays, loops automatically come into the picture. There are a variety of restrictions placed on the types of loops supported in the shading language. To boil it down to its simplest form, the `for` loops in WebGL must have an iteration count that is known at the compile time.

You should generally be cautious when using loops in WebGL. The basic restrictions are as follows:

- There must be only one loop iteration variable and it must be incremented or decremented using a simple statement (`i++`, `i--`, `i+=constant`, `i-=constant`)
- The stop condition must be a comparison between the loop index and the constant expression
- We must not change the value of the iterator in the loop

So, let's take a look at our shaders now.

The vertex shader

We declared a constant number of the positional light (`NUM_POSITIONAL_LIGHTS`). We declared a uniform array (`uLightPosition[NUM_POSITIONAL_LIGHTS]`) to hold the reference to positions of the lights. We declared a varying to hold a calculated light ray (`uLightRay[NUM_POSITIONAL_LIGHTS]`) per vertex. Then, we iterated over the array to calculate the light ray for each positional light. The code snippet for the

vertex shader is as follows:

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;
  uniform mat4 mVMatrix;
  uniform mat4 pMatrix;
  uniform mat4 nMatrix;
  varying vec3 transformedNormal;
  varying vec3 vertexPos;
  const int NUM_POSITIONAL_LIGHTS = 2;
  uniform vec3 uLightPosition[NUM_POSITIONAL_LIGHTS];
  varying vec3 uLightRay[NUM_POSITIONAL_LIGHTS];
  void main(void) {
    vec4 vertexPos4 =  mVMatrix  vec4(aVertexPosition, 1.0);
    vertexPos = vertexPos4.xyz;
    for(int i = 0; i < NUM_POSITIONAL_LIGHTS; i++){
      vec4 newLightPosition=mVMatrix  vec4(uLightPosition[i], 1.0);
      uLightRay[i]=vertexPos-newLightPosition.xyz;
    }
    transformedNormal = vec3(nMatrix  vec4(aVertexNormal,1.0));
    gl_Position= pMatrix vertexPos4;
  }
</script>
```

The fragment shader

The fragment shaders have two constants: one for the number of directional lights (`NUM_DIRECTIONAL_LIGHTS`) and the other is the number of positional lights (`NUM_POSITIONAL_LIGHTS`). Then, we declare three array uniforms (`uDirectionalDiffuseColor`, `uDirectionalSpecularColor`, and `uLightDirection`) for directional lights and two array uniforms (`uPositionalDiffuseColor` and `uPositionalSpecularColor`) for positional lights. We also declare a varying (`uLightRay`) whose values are passed from the vertex shader.

We start our calculation by initializing the `iDiffuse` and `iSpecular` components to `0`. Then, we iterate over the list of directional/positional lights to calculate the diffuse/specular component. We then add their corresponding values to the `iDiffuse` and `iSpecular` variables to get the sum total of all the lights in the scene. The code snippet for the fragment shader is as follows:

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
```

```
  varying vec3 transformedNormal;
  varying vec3 vertexPos;
  const int NUM_DIRECTIONAL_LIGHTS = 1;
  uniform vec3 uLightDirection[NUM_DIRECTIONAL_LIGHTS];
  const int NUM_POSITIONAL_LIGHTS = 2;
  varying vec3 uLightRay[NUM_POSITIONAL_LIGHTS];
  uniform vec3 uAmbientColor;
  uniform vec3 uDirectionalDiffuseColor[NUM_DIRECTIONAL_LIGHTS];
  uniform vec3 uDirectionalSpecularColor[NUM_DIRECTIONAL_LIGHTS];
  uniform vec3 uPositionalDiffuseColor[NUM_POSITIONAL_LIGHTS];
  uniform vec3 uPositionalSpecularColor[NUM_POSITIONAL_LIGHTS];
  uniform vec3 materialDiffuseColor;
  uniform vec3 materialAmbientColor;
  uniform vec3 materialSpecularColor;
  void main(void)  {
    vec3 normal=normalize(transformedNormal);
    vec3 eyeVector=normalize(-vertexPos);
    vec3 iAmbient=uAmbientColor*materialAmbientColor;
    vec3 iDiffuse=vec3(0.0,0.0,0.0);
    vec3 iSpecular=vec3(0.0,0.0,0.0);
    float specular = 0.0;
    for(int i = 0; i < NUM_DIRECTIONAL_LIGHTS ; i++){
      vec3 lightDirection = normalize(uLightDirection[i]);
      float directionalLightWeighting = max(dot(normal, -
lightDirection), 0.0);
      iDiffuse+=uDirectionalDiffuseColor[i] materialDiffuseColor
directionalLightWeighting;
      if(directionalLightWeighting>0.0)
      {
        vec3 halfDir = normalize(-lightDirection + eyeVector);
        float specAngle = max(dot(halfDir, normal), 0.0);
        specular = pow(specAngle, 4.0);

iSpecular+=uDirectionalSpecularColor[i]*materialSpecularColor*specu
lar;
      }
    }
    for(int  i = 0;i < NUM_POSITIONAL_LIGHTS ; i++){
      vec3 lightDirection = normalize(uLightRay[i]);
      float directionalLightWeighting = max(dot(normal, -
lightDirection), 0.0);
      iDiffuse+=uPositionalDiffuseColor[i] materialDiffuseColor
directionalLightWeighting;

      if(directionalLightWeighting>0.0)
      {
        vec3 halfDir = normalize(-lightDirection + eyeVector);
```

```
        float specAngle = max(dot(halfDir, normal), 0.0);
        specular = pow(specAngle, 4.0);

iSpecular+=uPositionalSpecularColor[i]*materialSpecularColor*specul
ar;
      }
    }
    vec3 iColor =iAmbient+ iDiffuse+iSpecular;
    gl_FragColor = vec4(iColor, 1.0);
  }
</script>
```

Implementing Light.js

We have added a new class to hold the components of light. Although the `Light` class has both `direction` and `position` variables defined, it is expected that we will use either of them in an object, as shown in the following code:

```
Light = function (  ) {
  this.specularColor=[0,0,0];
  this.diffuseColor=[0,0,0];
  this.ambientColor=[0,0,0];
  this.direction=[];
  this.position=[];
};
Light.prototype = {
  constructor: Light
}
```

Applying Lights.js

This class holds the objects of all the lights added to our game scene. We have defined a function `getDataByType()` to return the array of the component of the light. For example, if we invoke `lights.getDataByType("ambient")`, it will sum all the components (R, G, B) of the light ambient colors to return an array of three values (r, g, b). If we invoke `lights.getDataByType("position")` and if we have defined three positional lights, it will return an array with nine elements ([x0, y0, z0, x1, y1, z1, x2, y2, z2], the positions of the three lights in a single array). Similarly for the directional light, it will return a single array. If we invoke `lights.getDataByType("diffuse","position")`, it will return a diffuse component (R, G, B) of each positional light concatenated in a single array. The `Light` class is as follows:

```
Lights = function ( ) {
  this.lights=[];
```

```
};
Lights.prototype = {
  constructor: Lights,
  addLight:function(light){
    this.lights.push(light);
  },
  getDataByType:function(type,lightType){
    var sum=[];
    if(type=="position"){
      for(var i=0;i<this.lights.length;++i){
        if(this.lights[i].position.length>0){
          sum=sum.concat(this.lights[i][type])
        }
      }
    } else if(type=="direction"){
      for(var i=0;i<this.lights.length;++i){
        if(this.lights[i].direction.length>0){
          sum= sum.concat(this.lights[i][type])
        }
      }
    }
    else if(type=="ambientColor"){
      sum=[0,0,0];
      for(var i=0;i<this.lights.length;++i){
        sum[0]=sum[0]+this.lights[i].ambientColor[0];
        sum[1]=sum[1]+this.lights[i].ambientColor[1];
        sum[2]=sum[2]+this.lights[i].ambientColor[2];
      }
    }
    else{
      for(var i=0;i<this.lights.length;++i){
        if(this.lights[i][lightType].length>0){
          sum= sum.concat(this.lights[i][type])
        }
      }
    }
    return sum;
  }
}
```

Understanding the main code

We have added code in the `initShaders` function to get reference to the six uniform arrays. We have used this function in our shaders as follows:

```
function initShaders(){
```

```
  shaderProgram.uAmbientColor =
gl.getUniformLocation(shaderProgram, "uAmbientColor");
  shaderProgram.uLightDirection =
gl.getUniformLocation(shaderProgram, "uLightDirection");
  shaderProgram.uPositionalDiffuseColor =
gl.getUniformLocation(shaderProgram, "uPositionalDiffuseColor");
  shaderProgram.uPositionalSpecularColor =
gl.getUniformLocation(shaderProgram, "uPositionalSpecularColor");
  shaderProgram.uDirectionalDiffuseColor =
gl.getUniformLocation(shaderProgram, "uDirectionalDiffuseColor");
  shaderProgram.uDirectionalSpecularColor =
gl.getUniformLocation(shaderProgram, "uDirectionalSpecularColor");

  shaderProgram.uLightPosition =
gl.getUniformLocation(shaderProgram, "uLightPosition");
.............
}
```

We also have modified the `setLightUniform()` function to initialize our light objects and added each light to our `lights` object. We then initialized our uniforms' variable, which is returned from `lights.getDataByType`, with single dimensional arrays. In the following code, the WebGL function we used to initialize the uniforms is `gl.uniform3fv`. The following is the code for the `setLightUniform()` function:

```
function setLightUniform(){
  var light=new Light();
  light.ambientColor=[0.1,0.1,0.1];
  light.diffuseColor=[0.5,0.5,0.5];
  light.specularColor=[1.0,1.0,1.0];
  light.direction=[0,-1.25,-1.25];
  var light1=new Light();
  light1.diffuseColor=[lightColor[0],lightColor[1],lightColor[2]];
  light1.specularColor=[0.0,0.0,0.0];
  light1.position=[20.0,5.0,-100.0];
  var light2=new Light();
  light2.diffuseColor= [1.0,1.0,1.0];
  light2.specularColor=[0.0,0.0,0.0];
  light2.position=[20.0,5.0,200.0];
  var lights=new Lights();
  lights.addLight(light);
  lights.addLight(light1);
  lights.addLight(light2);

gl.uniform3fv(shaderProgram.uAmbientColor,lights.getDataByType("amb
ientColor"));
```

```
gl.uniform3fv(shaderProgram.uDirectionalDiffuseColor,lights.getData
ByType("diffuseColor","direction"));

gl.uniform3fv(shaderProgram.uDirectionalSpecularColor,lights.getDat
aByType("specularColor","direction"));

gl.uniform3fv(shaderProgram.uPositionalDiffuseColor,lights.getDataB
yType("diffuseColor","position"));

gl.uniform3fv(shaderProgram.uPositionalSpecularColor,lights.getData
ByType("specularColor","position"));
  gl.uniform3fv(shaderProgram.uLightPosition,
lights.getDataByType("position"));
  gl.uniform3fv(shaderProgram.uLightDirection,
lights.getDataByType("direction"));
}
```

Summary

This chapter focused on making our code capable to handle the loading and rendering of multiple objects. The other important aspect we covered in this chapter was the use of stacks to maintain the state of `mvMatrix` for object transformation operations and rendering.

The other major code update was addition of the rendering of our scene at a predefined frame rate. We also learned about positional lights and how to handle multiple lights in any scene. Also keep in mind that using lights in a scene is a GPU-intensive operation and if you have added a specular component to your light, then the calculations become more intensive. The overall performance of your game will depend on the intelligent use of lights in your scene.

We will play more with textures in our next chapter and will learn how to beautify the scenes even more.

Chapter 4. Applying Textures

In Chapter 3, *Loading the Game Scene*, we saw that our scene did not look very realistic. The reason was the use of one solid color on all the polygons of the object. Remember that we apply textures to surfaces to achieve the desired look and feel. This chapter covers the following topics on how to create, load, and apply textures to your scene:

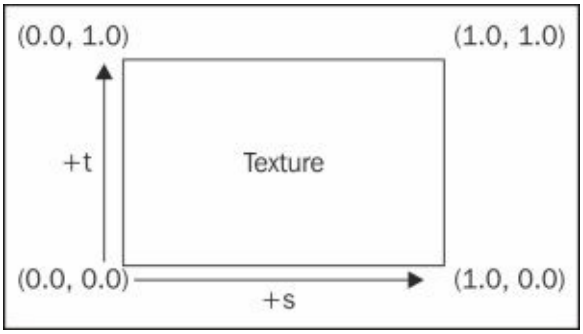
- Texturing basics
- Loading textures and using them in fragment shaders
- Texture filtering and wrapping
- Loading objects exported from Blender
- Mipmapping
- Cubemap textures

Texturing basics

There are mainly three types of textures: 2D textures (most common), Cubemap textures, and 3D textures. 3D textures are rarely used in gaming applications and are mostly used in volumetric effects such as light rays and realistic fog. We will not discuss 3D textures in this book as they are mostly procedural. To explain better, there is some software that can help you create or modify textures. Some of this software, such as PixPlant (http://www.pixplant.com/), is young and still evolving.

Understanding 2D textures and texture mapping

A 2D texture is a two-dimensional array of image data, a regular JPG or PNG image. The individual data elements of a texture are called texels. In 3D graphics, we use the term texels instead of image pixels. Texels are represented in different formats such as `gl.RGB`, `gl.RGBA`, `gl.LUMINANCE`, or `gl.ALPHA` along with their data types. Rendering with 2D textures requires a texture coordinate, which is an index into the image. Texture coordinates for 2D textures are given by a set of 2D coordinates, (s, t). The values for *s* and *t* range from 0.0 to 1.0. We can define values for (s, t) outside the range 0.0 to 1.0, but then the texture's behavior will be defined by the wrapping mode. We will cover the wrapping mode later in the chapter. The 2D coordinates, *s* and *t*, of a texture are shown in the following figure:



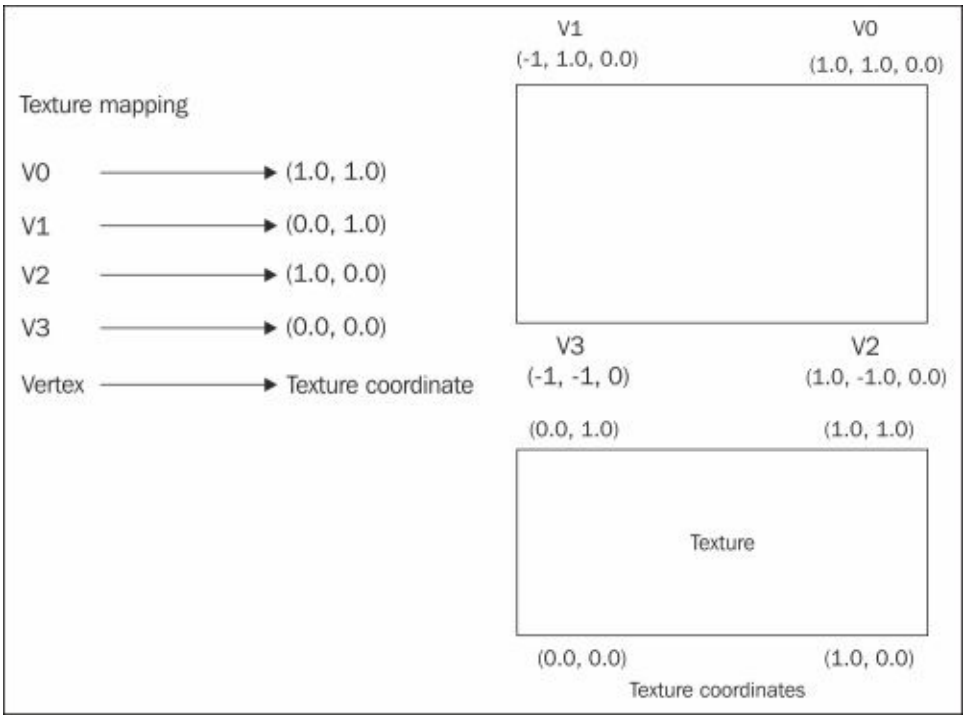
So, in a nutshell, each texel is indexed using a 2D index, called a texture coordinate, and each vertex of the mesh has a texture coordinate associated with it. The texture coordinates for each vertex in a polygon define what part of the image appears on that polygon.

So, let's keep to the philosophy of the book by learning the basics through code. The following code focuses on our new `initBuffers` function from Chapter 1, *Getting Started with WebGL Game Development*, in which we color our square with a texture. We earlier learned that we define a Vertex Buffer Object to store the vertex data in the GPU memory. The vertex data we used earlier consisted of vertex coordinates, colors per vertex, and normals per vertex. Here, we will create a new buffer for texture coordinates per vertex:

```
function initBuffers() {

  ...
```

```
  vertices = [
    1.0, 1.0, 0.0, //V0
    -1.0, 1.0, 0.0, //V1
    1.0, -1.0, 0.0, //V2
    -1.0, -1.0, 0.0 //V3
  ];

  indices = [0,2,3,0,3,1];

  ...
  var textureCoordinates = [
    1.0,  1.0,
    0.0,  1.0,
    1.0,  0.0,
    0.0,  0.0
  ];
...

}
```

In the preceding code, we have four vertices `V0` to `V3`, and we have the corresponding texture coordinates array. For each vertex, we have defined a vertex coordinate. The mapping is explained in the following figure:



We specify texture coordinates as fractions and not the exact image pixel size so that we keep our code independent of the texture size. The preceding figure explains that the vertex **V0**, the upper-right corner of the square, maps to **(1.0, 1.0)** of the texture coordinate and other coordinates are similarly mapped. This concept of mapping a vertex to texture coordinates is called texture mapping.

A mesh may be made up of many polygons, so a question may pop up in your mind: do we have to map each vertex to texture by hand? Well, no, we do not map each vertex by hand, as our 3D tools give us the texture to vertex mapping information and we only have to parse it.

Comprehending texture filtering

In texture filtering, the rendering engine determines the texture color for a texture-mapped pixel. The simplest algorithm selects the color from the nearby texel. This algorithm produces aliasing effects when the text is applied on different shapes, sizes, and angles. There are many algorithms evolved for anti-aliasing in order to minimize blurriness, shimmering, and blocking. The question is why does it even happen? Basically, during texture mapping, a texture lookup takes place to determine where each pixel center falls on the texture (texel to pixel mapping). As the object on which the texture is applied is at an arbitrary distance and orientation from the camera/viewer, one pixel on the screen might not correspond directly to a single texel. Some form of filtering has to be applied to determine the best color for the pixel.

There can be different types of correspondences between a pixel on the screen and the texel(s) it represents. Let's take an example of a 2D texture that is mapped on to a plane surface. At some viewing distance, the size of one screen pixel is exactly the same as one texel. If we move the plane closer to the camera/viewer, the texels appear larger than screen pixels and need to be scaled up. The process of scaling up the texture is known as **texture magnification**. If we move the plane away from the texture, each texel appears smaller than a pixel and so one pixel covers multiple texels. In this case, an appropriate color has to be picked up based on the covered texels via **texture minification**. We will learn multiple filtering techniques as we move along.

However, for now, we need to understand one concept: that each texel of the texture might not correspond to one pixel and that we apply filters to determine the best color for that pixel.

Loading textures

Let's first understand a few important WebGL API functions before diving into the code.

The `gl.pixelStorei` function sets pixel storage modes for `readPixels` and unpacks textures with `texImage2D` and `texSubImage2D`. The `gl.pixelStorei` function is declared as follows:

```
void gl.pixelStorei(GLenum pname, GLint param)
```

Let's see what each parameter of this function does:

- `pname`: This parameter specifies the pixel storage type; it must be either `gl.PACK_ALIGNMENT`, `gl.UNPACK_ALIGNMENT`, or `gl.UNPACK_FLIP_Y_WEBGL`
- `param`: This parameter specifies the Boolean value for the pack or unpack alignment

The `gl.pixelStorei` function is not used for a particular texture but is a global setting for all textures used in our current graphics context. We will use this function to flip the texture in our game, as shown in the following function call:

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

Normally, the texture coordinates in images start from the top-left corner. The *y* value increases as we move downwards, but WebGL starts from the bottom-left corner and the *y* value increases as we move upwards. The `gl.pixelStorei` function instructs WebGL to store the image with the *y* coordinate flipped.

Our gaming application must bind the texture object to be able to operate on it. Once the texture objects are bound, subsequent operations such as `gl.texImage2D` and `gl.texParameteri` affect the bound texture object. The following function makes the specified texture the current texture for further operations:

```
void gl.bindTexture(target, GLuint texture)
```

Let's see what each parameter of this function does:

- `target`: This parameter binds the texture object to the texture target, `gl.TEXTURE_2D` or `gl.TEXTURE_CUBE_MAP`

- `texture`: This parameter is the handle to the texture object

The primary function that is used for loading textures is `gl.texImage2D`. This function is very powerful for the purpose of gaming, and is declared as follows:

```
void gl.texImage2D(target, level, internalFormat, format, type,
const void* pixels)
```

Let's see what each parameter of this function does:

- `target`: This parameter specifies the texture target, either `gl.TEXTURE_2D` or one of the cubemap face targets (for example, `gl.TEXTURE_CUBE_MAP_POSITIVE_X`, or `gl.TEXTURE_CUBE_MAP_NEGATIVE_X`, among others).
- `level`: This parameter specifies which mip level to load. The base level is specified by `0` followed by an increasing level for each successive mipmap.
- `internalFormat`: This parameter specifies the internal format for the texture storage; this can be `gl.RGBA`, `gl.RGB`, `gl.LUMINANCE_ALPHA`, `gl.LUMINANCE`, or `gl.ALPHA`.
- `format`: This parameter specifies the format of the pixel data. The following symbolic values are accepted: `GL_RED`, `GL_RG`, `GL_RGB`, `GL_BGR`, `GL_RGBA`, and `GL_BGRA`.
- `type`: This parameter specifies the type of the incoming pixel data and can be `gl.UNSIGNED_BYTE`, `gl.UNSIGNED_SHORT_4_4_4_4`, `gl.UNSIGNED_SHORT_5_5_5_1`, or `gl.UNSIGNED_SHORT_5_6_5`.
- `pixels`: This parameter contains the actual pixel data for the image. The data must contain (width multiplied by height) the number of pixels with the appropriate number of bytes per pixel based on the format and type specification.

We are only using the simplest overloaded function as it is also capable of loading the texture data from the HTML `video` element. The second parameter, `level`, is the parameter used for loading mipmap levels which we will use in the next code packet.

To set texture parameters for the current texture unit, the `gl.texParameteri` function is used, which is declared as follows:

```
Void gl.texParameteri(GLenum target, GLenum pname, GLint param)
```

Let's see what each parameter of this function does:

- `target`: This parameter specifies the target texture, which must be one of the following: `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_1D_ARRAY`, `GL_TEXTURE_2D_ARRAY`, `GL_TEXTURE_RECTANGLE`, or `GL_TEXTURE_CUBE_MAP`
- `pname`: This parameter specifies the symbolic name of a single-valued texture parameter and can be one of the following: `GL_DEPTH_STENCIL_TEXTURE_MODE`, `GL_TEXTURE_BASE_LEVEL`, `GL_TEXTURE_COMPARE_FUNC`, `GL_TEXTURE_COMPARE_MODE`, `GL_TEXTURE_LOD_BIAS`, `GL_TEXTURE_MIN_FILTER`, `GL_TEXTURE_MAG_FILTER`, `GL_TEXTURE_MIN_LOD`, `GL_TEXTURE_MAX_LOD`, `GL_TEXTURE_MAX_LEVEL`, `GL_TEXTURE_SWIZZLE_R`, `GL_TEXTURE_SWIZZLE_G`, `GL_TEXTURE_SWIZZLE_B`, `GL_TEXTURE_SWIZZLE_A`, `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, or `GL_TEXTURE_WRAP_R`
- `param`: This parameter specifies the value of `pname`

The `gl.texParameteri` function sets the texture filtering mode for each texture used in our application. It operates on the currently bound texture and sets the filter mode for the texture. This also means that for every texture we load, we will have to specify the filter mode for it.

We will learn all filter modes as we move but for now, we will be using the nearest-neighbor interpolation method. Take a look at the following code:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

Nearest-neighbor interpolation is the fastest and crudest filtering method. It simply uses the color of the texel closest to the pixel center for the pixel color. However, the rendering of textures is not as good as expected. It results in a large number of artifacts such as blockiness in case of magnification and aliasing/shimmering in minification.

The preceding two lines of code set the nearest-neighbor interpolation filter for both magnification and minification.

We will introduce a new data type in the ESSL code, `sampler2D`, to reference the texture data in the shader and texture lookup functions.

A new data type – sampler

A sampler data type can be `sampler1D`, `sampler2D`, and `sampler3D`. A variable of the sampler can only be defined in one of two ways. It can be defined as a function parameter or as a uniform variable, shown as follows:

```
uniform sampler2D texture1;
void Function(in sampler2D myTexture);
```

Samplers do not have a value. They cannot be set by expressions in a shader. They can only be passed to a function as an `in` parameter. Putting it together, a sampler value can be set in the shader code.

The sampler can only be used as a parameter in one of the GLSL standard library's texture lookup functions. These functions access the texture referred to by the sampler. They take a texture coordinate as parameters.

We will cover only two functions for now, `texture` and `textureOffset`. The `texture` function is declared as follows:

```
vec texture(sampler sampler, vec texCoord);
```

The process of fetching data from a texture, at a particular location, is called sampling. This function returns the samples of the texture associated with the sampler, at the location `texCoord`. The size of the `vec` type of `texCoord` depends on the dimensionality of the sampler. A 1D sampler takes a `float` size whereas a 2D sampler takes a `vec2` size.

```
vec textureOffset(sampler sampler, vec texCoord, vec offset);
```

We can add a texel offset to the texture coordinates, sampled with texture functions. This is useful for sampling from a collection of images, all on a single texture.

Applying a texture to the square

In our code example, we will simply add a texture to the square we rendered in Chapter 1, *Getting Started with WebGL Game Development*. We will now just summarize the changed code snippets that we added to render the square with a texture instead of a predefined color. Open the `04-SquareWithTexture.html` file in your favorite text editor and review the following changes:

1. We added the texture map array to the `initBuffers` function.
2. We created a buffer, made the buffer the active buffer using `bindBuffer`, and allocated memory for the buffer. Note that the texture coordinates are defined for each vertex, as shown in the following code snippet:

```
function initBuffers() {
  ...
  verticesTextureBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, verticesTextureBuffer);

  var textureCoordinates = [
    1.0,  1.0,
    0.0,  1.0,
    1.0,  0.0,
    0.0,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(textureCoordinates), gl.STATIC_DRAW);
}
```

The vertex shader

A new attribute, `aTextureCoord`, is defined to hold the texture coordinates assigned from the flow code. A new varying, `vTextureCoord`, is defined to pass the texture coordinate to the fragment shader. Remember, the only way to pass the bind data from the flow code to the shader variable is by defining an attribute and to pass the data from the vertex shader to the fragment shader is by defining a uniform. Then, we assign the value of `aTextureCoord` to `vTextureCoord` so that this value can be used by the fragment shader. All we are doing is accepting the texture coordinates as a per-vertex attribute and passing it straight out in a varying variable, as shown in the following code snippet:

```
<script id="shader-vs" type="x-shader/x-vertex">
  ...
  attribute vec2 aTextureCoord;
```

```
  varying highp vec2 vTextureCoord;
  void main(void) {
    ...
    vTextureCoord = aTextureCoord;
  }
</script>
```

The fragment shader

In the following code, we declare two variables; one is a `sampler` that holds the reference to texture that we loaded in the main control flow. The second variable is `vec2varyingvTextureCoord`, which is passed from the vertex shader. The important function is `texture2D`, which accesses the sampler at the texel defined by the `vTextureCoord` variable and gets the color value for that texel.

```
<script id="shader-fs" type="x-shader/x-fragment">
  uniform sampler2D uSampler;
  varying highp vec2 vTextureCoord;
  void main(void) {
    gl_FragColor = texture2D(uSampler, vec2(vTextureCoord.s,
vTextureCoord.t));
  }
</script>
```

In the `initShaders` function, the new variable we have added is `textureCoordAttribute`. It references the `aTextureCoord` attribute of the vertex shader, as shown in the following code:

```
function initShaders() {
  ...
  shaderProgram.textureCoordAttribute =
gl.getAttribLocation(shaderProgram, "aTextureCoord");

  gl.enableVertexAttribArray(shaderProgram.textureCoordAttribute);
  ...
}
```

The `createTexture()` function creates the texture reference. To actually create the texture object itself, you must bind it to the current texture context. We do that after we load the texture. We create the image object and once the image is loaded, the `onload` event invokes `handleTextureLoaded`. We then bind the texture to actually create the object using the `bindTexture` function. This function also makes the texture object the current active texture buffer. The `texImage2D` function loads the texture in GPU memory and associates it with the

active texture object. The `texImage2D` function loads the `img` object at level of detail `0` (mipmap level), and we specify the input texture format, `gl.RGBA`, and the storage format. The next call, `texParameteri`, defines the filter mode for our texture. We use the nearest-neighbor interpolation algorithm to filter for magnification and minification. Then, we invoke the `drawscenedrawScene` function, as shown in the following code:

```
function initTextures() {
  texture = gl.createTexture();
  image = new Image();
  // Assigning onLoad Event before setting the source, since the
  //onload will never be invoked if we assign the source earlier.
  image.onload = function() {
    handleTextureLoaded(image, texture);}
  image.src = "cubetexture.png";
}

function handleTextureLoaded(img, texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, img);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);
  gl.bindTexture(gl.TEXTURE_2D, null);
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);
  drawScene();
}
```

Note

In the `handleTextureLoaded` function, we invoke `drawScene` after the texture is loaded. That is not what we will do in our real game. We will invoke `drawScene` at a particular frame rate which means it will be invoked continually. Also, we will load multiple textures, and we can wait for all textures to get loaded before we start rendering.

The `drawScene` function is modified to initialize the shader variables. In the following code, we have added two new variables, one attribute, `aTextureCoord`, in the vertex shader and the other uniform, `uSampler`, in the fragment shader. The first line makes `verticesTextureBuffer` the current buffer by the `bindBuffer` call. Then, the `vertexAttribPointer` function is invoked to map

the buffer to the shader variable, `aTextureCoord`, through its reference `textureCoordAttribute`. The next lines are interesting. WebGL can deal with up to 32 textures during any given call to functions such as `gl.drawElements`. These textures are numbered from `TEXTURE0` to `TEXTURE31`. The first line tells WebGL that texture `0` is the one we loaded, and the second line defines the current texture object. The third line passes the value `0` to a shader uniform. In other words, the uniform shader, `uSampler`, will hold the reference to the first texture, as shown in the following code:

```
function drawScene() {
  ...
  gl.bindBuffer(gl.ARRAY_BUFFER, verticesTextureBuffer);
  gl.vertexAttribPointer(shaderProgram.textureCoordAttribute, 2,
gl.FLOAT, false, 0, 0);

  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.uniform1i(gl.getUniformLocation(shaderProgram, "uSampler"),
0);
  ...
}
```

Texture wrapping

As seen in the previous example in the *Applying a texture to the square* section, let's first take a quick look at the texture mapping coordinates.

```
var textureCoordinates = [
  1.0,  1.0,
  0.0,  1.0,
  1.0,  0.0,
  0.0,  0.0
];
```

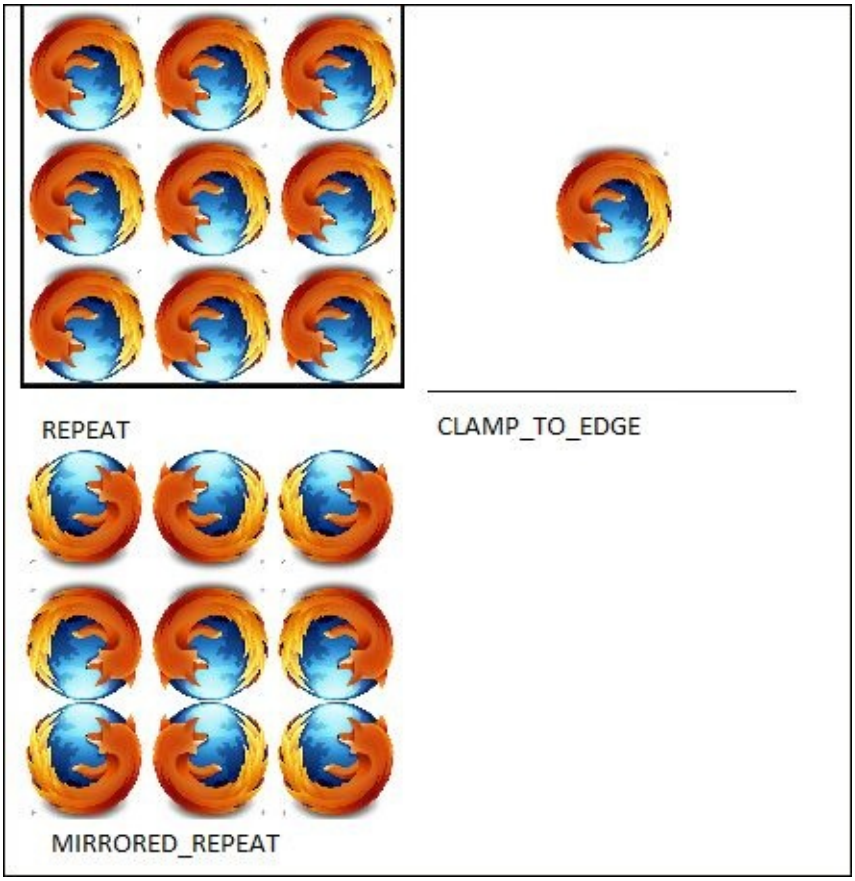
The texture coordinates lie in the range 0.0 to 1.0. However, imagine if we use values like -1 to 2.0 or 1.5 to 2.0. Now, these values lie outside the previous range. Which texel from the texture should WebGL pick? Let's think for a second and evaluate our options. If the *s* value goes outside the range, we could say that if *s > 1.0*, then *s = 1.0* and if *s < 0.0*, then *s = 0*. Here, we are clamping, which means that we are always picking the border values if the value goes outside the range. Or, if we say the texture is cylindrical and *s = 1.2*, then the integer value is dropped and we get the value 0.2. So, texture wrapping describes which texel to pick if the values lie outside the range [0.0, 1.0]. The wrapping mode is specified separately for both *s* and *t* values.

The following table lists the different wrapping modes that we can use in the `texParameteri` API call:

Wrapping mode	**Description**
`CLAMP_TO_EDGE`	If the sorted values exceed the range 0 to 1, then the border values will be taken.
`REPEAT`	This is the default texture wrapping mode. It gives a tiled look to the texture, just like a brick wall. The integer of the *s* or *t* value is ignored, for example: (0.5, 1.6) becomes (0.5, 0.6).
`MIRRORED_REPEAT`	In this mode, if the integer part of the real number (*s* or *t*) is even, the integer part is ignored. If the integer part is odd,

	even, the integer part is ignored. If the integer part is odd, then the integer part is ignored and the fraction is subtracted from 1, for example: (2.4, 3.2) becomes (0.4, 0.8) and (3.2, 2.4) becomes (0.8, 0.6).

The following figure shows the outputs of the different wrapping modes:



Testing the texture wrapping mode

The code in the `04-SquareWithTextureWrapping.html` file gives us options to try texture wrapping for both *s* and *t* values separately. Open the file in your favorite text editor. We will see the various changes that we made in the code.

The HTML

We have added two select boxes to list different wrapping modes for *s* and *t* values, with IDs `sclamp` and `tclamp` respectively, as shown in the following code snippet:

```
<div>
  <label for="sclamp">Clamping Mode S</label>
  <select id="sclamp" name="sclamp">
    <option value="REPEAT">REPEAT</option>
    <option value="CLAMP_TO_EDGE">CLAMP_TO_EGDE</option>
    <option value="MIRRORED_REPEAT">MIRRORED_REPEAT</option>
  </select>
  <label for="tclamp">Clamping Mode T</label>
  <select id="tclamp" name="tclamp">
    <option value="REPEAT">REPEAT</option>
    <option value="CLAMP_TO_EDGE">CLAMP_TO_EGDE</option>
    <option value="MIRRORED_REPEAT">MIRRORED_REPEAT</option>
  </select>
</div>
```

The event handlers

We use the jQuery library to help with programming the web page. Here are the on-change event handlers for both select boxes:

```
<script type="text/javascript" src="js/jquery.js"></script>
function start() {
  ...
  $("#sclamp").change(redrawWithClampingMode);
  $("#tclamp").change(redrawWithClampingMode);
  ...
}
```

The redrawWithClampingMode function

We make the texture as the current/active texture by using the `bindTexture` API call. We retrieve the selected value of the select boxes using jQuery API

selectors. Then, depending on the selected value, we set the value of the parameter `gl.TEXTURE_WRAP_S` or `gl.TEXTURE_WRAP_T` using the `texParameteri` API call for the different wrapping mode. Then, we invoke the `drawScene` function after clearing the current texture buffer, as shown in the following code snippet:

```
function redrawWithClampingMode() {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  var sValue=$("#sclamp option:selected").val();
  var tValue=$("#tclamp option:selected").val();
  switch(sValue) {
    case"REPEAT":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.REPEAT);
      break;
    case"CLAMP_TO_EDGE":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
      break;
    case"MIRRORED_REPEAT":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.MIRRORED_REPEAT);
      break;
  }
  switch(tValue) {
    case"REPEAT":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.REPEAT);
      break;
    case"CLAMP_TO_EDGE":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
      break;
    case"MIRRORED_REPEAT":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.MIRRORED_REPEAT);
      break;
  }
  gl.bindTexture(gl.TEXTURE_2D, null);
  drawScene();
}
```

Exporting models from Blender

The most exciting part is here. In this section, we will export objects with textures from Blender. However, first we need to understand the structure of the newly exported object and the material file. Let's go through the exported cube with textures. Open the `Box.obj` file present in the `model` directory in your favorite text editor. The content of the file is as follows:

```
# Blender v2.67 (sub 0) OBJ File: ''
# www.blender.org
mtllib Box.mtl
o Box
v -0.738564 -0.738564 0.738564
v 0.738564 -0.738564 0.738564
v 0.738564 0.738564 0.738564
v -0.738564 0.738564 0.738564
v 0.738564 0.738564 -0.738564
v -0.738564 0.738564 -0.738564
v 0.738564 -0.738564 -0.738564
v -0.738564 -0.738564 -0.738564
vt 0.375000 0.000000
vt 0.625000 0.000000
vt 0.625000 0.250000
vt 0.375000 0.250000
vt 0.625000 0.500000
vt 0.375000 0.500000
vt 0.375000 0.750000
vt 0.625000 1.000000
vt 0.375000 1.000000
vt 0.625000 0.750000
vt 0.875000 0.000000
vt 0.875000 0.250000
vt 0.125000 0.000000
vt 0.125000 0.250000
vn 0.000000 -0.000000 1.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 -1.000000 -0.000000
vn 1.000000 0.000000 0.000000
vn -1.000000 -0.000000 -0.000000
g Box_Box_initialShadingGroup.004
usemtl initialShadingGroup.004
s off
f 1/1/1 2/2/1 3/3/1
f 4/4/2 3/3/2 5/5/2
```

```
f 6/6/3 5/5/3 8/7/3
f 2/8/4 1/9/4 7/10/4
f 2/2/5 7/11/5 5/12/5
f 8/13/6 1/1/6 6/14/6
f 4/4/1 1/1/1 3/3/1
f 6/6/2 4/4/2 5/5/2
f 5/5/3 7/10/3 8/7/3
f 1/9/4 8/7/4 7/10/4
f 3/3/5 2/2/5 5/12/5
f 1/1/6 4/4/6 6/14/6
```

Let's understand this file format. The `Box.obj` file has six types of elements defined, which are as follows:

- `mtllib`: This element defines the name of the material file attached. We will cover this in more detail shortly.
- `o`: This element defines the objects exported. Remember that we can have many objects in a single OBJ file. In our case, we have a single object, that is, `Box`.
- `v`: This element defines the vertices in the object. This file shows eight vertices for a cube. The exporter tries to reduce redundancy of vertices.
- `vt`: This element defines the vertex texture coordinates (*x*, *y* values) of the texture listed in the `Box.mtl` file. The two values corresponding to `vt` are called UV coordinates and are listed in each row. This is the reason that we also refer to texture mapping as UV mapping. The `Box.obj` file defines 15 UV coordinates.
- `vn`: This element defines the normals of the cube. It defines eight normals.
- `s`: This element allows us to enable or disable smooth shading. The value, `off`, denotes that smooth shading has been disabled.
- `f`: This element defines each vertex of a face/primitive in the v/n/t format depicted as follows:

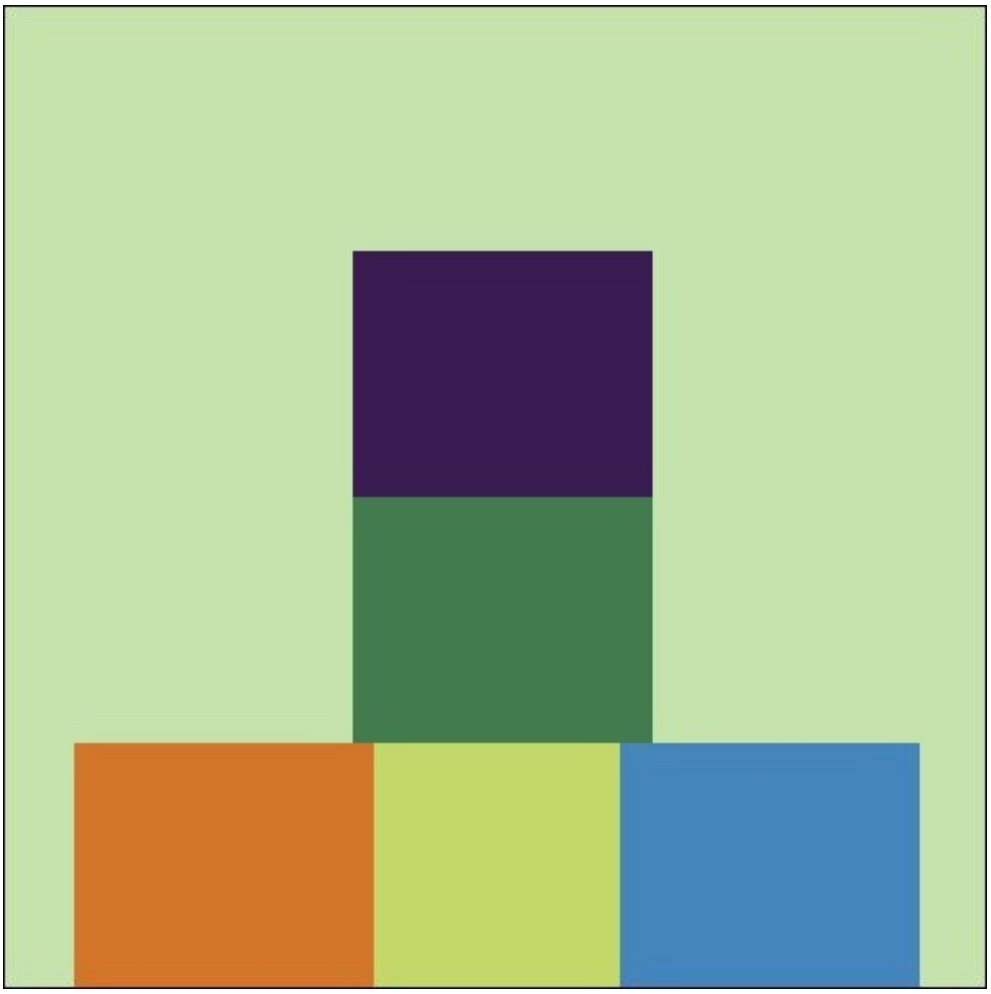
  ```
  f v/n/t v/n/t v/n/t
  f 1/1/1 2/2/1 3/3/1
  ```

  These three values (v, n, and t) show that each face of the `Box` object is defined by three vertices, or a triangle. The value `1/1/1` means that you pick the first vertex from the vertices list, the first normal from the normal list, and the first texture coordinate from the texture coordinates list. In short, we can say, each vertex of a face is defined by the vertex index, normal index, and texture coordinate index (v/n/t). The relation between the vertex and the texture coordinate is thus defined in the face definition.

Now, open `Box.mtl` in your favorite text editor. The content of the file is as follows:

```
# Blender MTL File: 'None'
# Material Count: 1
newmtl initialShadingGroup.004
Ns 96.078431
Ka 0.000000 0.000000 0.000000
Kd 0.000000 0.000000 0.000000
Ks 0.500000 0.500000 0.500000
Ni 1.000000
d 0.000000
illum 2
map_Kd boxDiffuse.jpg
```

The `Ka` element defines the ambient color, `Kd` defines the diffuse color, and `Ks` defines the specular color. The `map_kd` element defines the name of the file with the texture. The texture mapping is defined in the `Box.obj` file. The following figure shows the texture map to texture a cube:



Converting Box.obj to Box.json

To import the `Box.obj` file with JavaScript in our code, we need to convert it to the JSON format as shown in the following command-line code:

```
python ./convert_obj_three.py -i ./Box.obj -o ./Box.json
```

Understanding the JSON file with UV coordinates

Let's understand our JSON file format with texture coordinates. Open `Box.json` in your favorite text editor. The content of the file is as follows:

```
{
  "metadata" :
  {
    "formatVersion" : 3.1,
    "sourceFile"    : "Box_Blender.obj",
    "generatedBy"   : "OBJConverter",
    "vertices"      : 8,
    "faces"         : 12,
    "normals"       : 6,
    "colors"        : 0,
    "uvs"           : 14,
    "materials"     : 1
  },
  "scale" : 1.000000,
  "materials": [ {
    "DbgColor" : 15658734,
    "DbgIndex" : 0,
    "DbgName" :"initialShadingGroup.004",
    "colorAmbient" : [1.0, 1.0, 1.0],
    "colorDiffuse" : [1.0, 1.0, 1.0],
    "colorSpecular" : [0.5, 0.5, 0.5],
    "illumination" : 2,
    "mapDiffuse" : "boxDiffuse.jpg",
    "opticalDensity" : 1.0,
    "specularCoef" : 96.078431,
    "transparency" : 0.0
  }],
  "vertices": [...],
  "morphTargets": [],
  "morphColors": [],
  "normals": [0,-0,1,0,1,0,0,0,-1,0,-1,-0,1,0,0,-1,-0,-0],
  "colors": [],
  "uvs":
[[0.375,0,0.625,0,0.625,0.25,0.375,0.25,0.625,0.5,0.375,0.5,0.375,0
.75,0.625,1,0.375,1,0.625,0.75,0.875,0,0.875,0.25,0.125,0,0.125,0.2
5]],
"faces":
[42,0,1,2,0,0,1,2,0,0,0,42,3,2,4,0,3,2,4,1,1,1,42,5,4,7,0,5,4,6,2,2
,2,42,1,0,6,0,7,8,9,3,3,3,42,1,6,4,0,1,10,11,4,4,4,42,7,0,5,0,12,0,
13,5,5,5,42,3,0,2,0,3,0,2,0,0,0,42,5,3,4,0,5,3,4,1,1,1,42,4,6,7,0,4
,9,6,2,2,2,42,0,7,6,0,8,6,9,3,3,3,42,2,1,4,0,2,1,11,4,4,4,42,0,3,5,
```

```
0,0,3,13,5,5,5]
}
```

We see three new entries in this JSON file. Apart from the regular vertices and normals array, we notice a new `uvs` array. It is a simple one dimensional array with each pair defining one texture or one UV coordinate. Also, notice the `materials` array, which has one material defined, with a new entry of `mapDiffuse` that refers to the diffuse map image, `boxDiffuse.jpg`.

The `faces` array looks a little different now from what we saw in Chapter 2, *Colors and Shading Languages*. It starts with a value of `42`. If we remember from Chapter 2, *Colors and Shading Languages*, the first value of the array defines the structure of the faces of each cube, so let's dive deep into it.

Let's first convert 42 into binary, which is 00101010. Starting from the last digit, we have listed what each bit means:

Digit	Denotes	Meaning
0	`isQuad`	The digit value `0` means it is not a quad, but a triangle. Three elements will give vertex indices of the face.
1	`hasMaterial`	The digit value `1` means we have a material associated with the polygon. Hence, the fourth value would give us the material index.
0	`hasFaceUV`	The JSON format stores the UV coordinate per face or per vertex. If it is per face, we call it a face UV. This essentially conveys that all the vertices of the face map to the same UV coordinate.
1	`hasFaceVertexUV`	The digit value of `1` means that the next two values will give the texture coordinates (UV).
0	`hasFaceNormal`	The digit value of `0` means that there is no face normal.

1	`hasFaceVertexNormal`	The digit value of `1` means that the next three values will give normal values for the vertices.
0	`hasFaceColor`	The digit value of `0` means that there is no face color.
0	`hasFaceVertexColor`	The digit value of `0` means that there is no face vertex color.

So let's sum it up; three values for the vertex index, one value for the material index, two values for the texture index, and three values for the normal index. A total of one (to define the structure) plus nine values will define a face for this particular JSON file.

Parsing UV coordinates from the JSON file

Now, we will walk you through the code to show the changes that we need to perform to load the JSON file with a texture. But, first let's understand the biggest challenge we are going to face to load the JSON file in order to have the minimum memory footprint.

The challenge and the algorithm

First, let's review the basic principles of WebGL texture mapping:

- Each texture coordinate is mapped to a vertex and vice versa. So, if we have eight vertices defined in the vertex buffer, we need eight texture coordinates. If we have 16 vertices defined in the vertex buffer, we need 16 texture coordinates. Take a look at the previous code snippets of this chapter and you will notice we had four UV values for four vertices of the square. This applies to all vertex attributes, such as colors and normals, among others.
- We used the index buffer so that we do not have to repeat vertices in the vertex buffer. We did this so that the memory we assign for the vertex buffer in the GPU is of the smallest footprint. However, there is no rule that says that the vertices cannot be redundant. We can repeat vertices, but then we would have to change the indexing in the `indices` array.
- There is no relation of the texture coordinates with the indices defined.

Why are we discussing this? Notice in the `Box.json` file that you have eight vertices but 15 texture coordinates. This will happen with any visual designing tool, be it Blender, Maya, or 3ds Max, the number of vertices will never match the number of texture coordinates. The export tools will always generate a minimum number of vertices and texture coordinates and their relation will be expressed in the face information such as the `Box.obj` file. We have more UV coordinates than vertex coordinates because a vertex can be shared between many faces and we can have different UV coordinates for each face. This is explained and implemented in depth in the upcoming sections.

Again, texture coordinates are relative to the vertex data and not the index data. The indexing is independent of texturing.

Suppose you have the following vertex array made of three vertices (three components each):

```
var vertices = [x0, y0, z0, x1, y1, z1, x2, y2, z2];
```

Also, there is a texture coordinate array made of three coordinates (two components each):

```
var t-co-ordinates = [s0, t0,    s1, t1,    s2, t2];
```

We associate the vertex 0 (`x0, y0, z0`) with the texture coordinate 0 (`s0, t0`). The index data is a way of specifying the vertices. In our previous example, if we would like to render the triangle twice, we would specify an index array as follows:

```
var indices[] = {0, 1, 2, 0, 1, 2};
```

The index data is independent of texture coordinates or other vertex attributes such as colors and normals, among others. Hence, we do not bind texture coordinates to the index data.

So, if we have a situation like in our case, where the number of texture coordinates does not match the number of vertices, then we have to create redundant vertex information or texture information and re-arrange our indices.

Now, we need to prepare our vertex array, index array, normal array, and texture coordinates array from the face information before making the `drawElements` call.

Revisiting vertices, normals, and the indices array

Let's summarize how we created our vertices, normals, and the indices arrays so far. We were simply copying our vertices from the JSON object to the vertices array.

The `parseJSON` function in the `parseJSON.js` file, present in the `primitive` directory, was reading the `Box.json` file and creating face objects, as shown in the following code snippet:

```
Face = function (a, b, c, normal, color, materialIndex) {
  this.a = a;
  this.b = b;
  this.c = c;
  this.normal =  normal ;
  this.vertexNormals = [ ];
  this.vertexColors = color instanceof Array ?color : [];
  this.colorIndex = color;
  this.vertexTangents = [];
  this.materialIndex = materialIndex !== undefined ? materialIndex
: 0;
};
```

Each face object had the structure, shown in the preceding code snippet, with `a`,

`b`, and `c` as the indices to the vertices for that face. We simply iterated over the `faces` array of the `Geometry` class and copied the `a`, `b`, and `c` values of each face object into our indices array using the `indicesFromFaces` function of our `Geometry` class (defined in `primitive/Geometry.js`), as shown in the following code snippet:

```
indicesFromFaces: function () {
  for(var i=0; i<this.faces.length; ++i) {
    this.indices.push(this.faces[i].a);
    this.indices.push(this.faces[i].b);
    this.indices.push(this.faces[i].c);
  }
}
```

For normals, we wrote a function, `calculateVertexNormals`, in the `Geometry` class. We did not use the `normal` array from the `Box.json` file because normals are associated with vertices and not indices or faces. The `Box.json` file gave a normal for each vertex with respect to that face. Hence, if a vertex was being used in two faces, then we had two normals for the same vertex and to get the final normal for that vertex, we had to add them up. However, we decided not to use that information and we calculated our normal array from indices and vertices.

Restructuring for texture coordinates

First, let's study the changes we need to perform to read the UV information of faces in our `parseJSON.js` file:

```
fi = geometry.faces.length;
if ( hasFaceUv ) {
  for ( i = 0; i < nUvLayers; i++ ) {
    uvLayer = data.uvs[ i ];
    uvIndex = faces[ offset ++ ];
    geometry.faceUvs[ i ][ fi ] = uvIndex;
  }
}

if ( hasFaceVertexUv ) {
  for ( i = 0; i < nUvLayers; i++ ) {
    uvLayer = data.uvs[ i ];
    uvs = [];
    var aVertexIndices=["a","b","c","d"];
    for ( j = 0; j < nVertices; j ++ ) {
      uvIndex = faces[ offset ++ ];
      uvs[ aVertexIndices[j] ] = uvIndex;
```

```
      uvs[ uvIndex[j]]  = uvIndex;
    }

    geometry.faceVertexUvs[ i ][ fi ] = uvs;
  }
}
```

The first `if` condition in the preceding code checks whether the JSON has the face's UV coordinates, then stores the index to the `uvs` array in the corresponding face index. Hence for each face, we will have a UV coordinate. This case will not prove untrue for an OBJ file as the OBJ format stores UV coordinates per face vertex. However, there are other formats where this case might be helpful.

The second `if` condition checks whether the JSON file has face UV coordinates per vertex, then stores the index of the texture coordinate for each vertex in a `uvs` array. Then, it stores the UV index array in the corresponding face index of the `faceVertexUvs` array.

The outer `i` loop of the second `if` condition is for the material index. An object can have multiple materials. In our case, we have a single material, but in most cases, you might find multiple materials for a geometry. Hence, `faceUvs[materialIndex][faceIndex]` is a double dimensional array declared in the `Geometry` class and `faceVertexUvs[material index][faceIndex][vertexIndex]` is a three dimensional array.

Algorithm one to create new arrays

Let's take a look at the `verticesFromFaceUvs` function from `Geometry.js`, which is present in the `primitive` directory. We have commented out the function since we do not use it in our examples—it is just for our reference.

The `verticesFromFaceUvs` function takes `vertices`, `uvs`, and `materialIndex` as parameters. The values of `vertices` and `uvs` are derived from the JSON object. They are redundant data. The actual arrays from the JSON file are passed into the function, as shown in the following code snippet:

```
/* verticesFromFaceUvs: function(vertices, uvs, materialIndex) {
  var vertexVectors = []; // will hold the redundant indexes to the
vertex array
  var redundantVertexVectors = []; // Will hold the redundant
indexes to the uv array
  var redundantUVs[]; // Create vector vertex from vertices for
easy indexing
  for(var i=0; i<vertices.length; i=i+3) {
```

```
    var vector = vec3.fromValues(vertices[i], vertices[i+1],
vertices[i+2]);
    vertexVectors.push(vector);
  }
  // One faceVertexUV corresponds to one face
  for(var i=0; i<this.faceVertexUvs[materialIndex].length; ++i) {
    var face=this.faces[i];  // Pick one face

    var textureIndices = this.faceVertexUvs[materialIndex][i]; //
Pick the corresponding vertexUV map
    var aVertexIndices = ["a", "b", "c"];
    for(var i=0; i<aVertexIndices.length; ++i) {
      var aVertexIndex = aVertexIndices[i];
      // For each vertex in face, copy the vertex to the vertex
redundant array.
      redundantVertexVectors.push(face[aVertexIndex]);
      // For each vertex in face, copy the vertexUV map to the uv
redundant array.
      redundantUVs.push(textureIndices[aVertexIndex]);
    }
  }
  for(var i=0; i<redundantVertexVectors.length; ++i) {
    var vector=vertexVectors[redundantVertexVectors[i]];
    // Copy X value of the vertex to vertices

    this.vertices.push(vector[0]); // Copy Y value of the vertex to
vertices

    this.vertices.push(vector[1]); // Copy Z value of the vertex to
vertices

    this.vertices.push(vector[2]); // Copy s value from the UV
array. Since uv=[s0, t0, s1, t1, s2, t2  ]
    this.uvs.push(uvs[redundantUVs[i]*2]);
    this.uvs.push(uvs[redundantUVs[i]*2+1]);
    this.indices.push(i+1); // indices=[1,2,3, 4,5,6, 7,8,9, ...]
  }
}, */
```

However, while it is the simplest approach you could take, it's been commented out and is there simply for our reference. It is not really the best approach as it is inefficient. The first `for` loop iterates over non-redundant vertices information and clubs triplets to create a single vector and pushes the vector on the `vertexVectors` array. We do this because indices point to a coordinate and not its individual *x*, *y*, or *z* components. The first `for` loop is shown as follows:

```
for(var i=0; i<vertices.length; i=i+3) {
  var vector = vec3.fromValues(vertices[i], vertices[i+1],
vertices[i+2]);
  vertexVectors.push(vector);
}
```

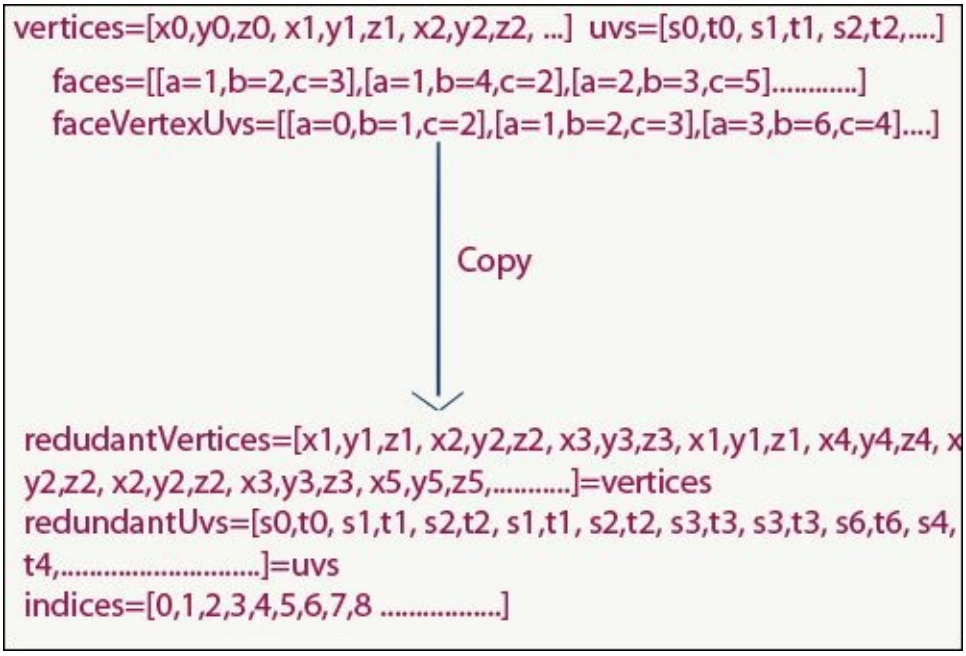
The second `for` loop iterates over the `faceVertexUvs` array. Remember that each element of the `faceVertexUvs` array corresponds to a face element of the `faces` array in the `Geometry` class. Hence, you could iterate over either of them, as shown in the following code snippet:

```
var face=this.faces[i]; // Pick one face
var textureIndices = this.faceVertexUvs[materialIndex][i]; // Pick
the corresponding vertexUV map
```

Retrieve one face element from the `faces` array and one element from the `faceVertexUv` array. The face element would contain indices to the vertices array that make up a polygon, in our case triangle. Each UV element (textureIndices) would contain indices to the UV array for each vertex. Hence, we iterate over these elements to retrieve the value for each vertex denoted by `a`, `b`, and `c`, and store the value for each vertex in the `redundantVertex` and `redundantUvs` arrays. We copy each vertex index and UV index into their corresponding redundant array as follows:

```
redundantVertexVectors.push(face[aVertexIndex]); // For each vertex
in face, copy the vertex to the vertex redundant array.
redundantUVs.push(textureIndices[aVertexIndex]); // For each vertex
in face, copy the vertexUV map to the uv redundant array.
```

So, basically we copied each vertex and UV coordinate in an order as specified in the `faces` and `faceVertexUv` arrays. This means we would not even need the `indices` array and can use the `drawArrays` call without any indices. On the other hand, we can create the `indices` array by simply storing elements starting from zero to the number of faces multiplied by 3. Here, we are repeating each possible vertex and UV coordinate as per the faces. As mentioned earlier, this would be a pretty inefficient approach, so this code is commented out. Thus, if an object had 1200 faces, then our vertex array would be *1200*3* elements and the UV array would have *1200*2* elements. Take a look at the following figure:



Note

Algorithm one is not the right way to proceed. It is good for objects such as a plane where the vertex information might not be redundant and indices might not be that valuable, but for most objects in games, vertices are reused. Hence, using this algorithm is not a very bright idea and so, we have commented out the function.

Algorithm two to create new arrays

In this algorithm, let's first understand the problem again. We need the same number of vertices as the UVs before we invoke the `drawElements` function call. So, we know that each `face` has a vertex index and `faceVertexUvs` has a UV index from that geometry. So, we have a relation between UV coordinates and vertices as each `faceVertexUv` element corresponds to a `face` element.

Our new algorithm is pretty simple. If a vertex index is pointing to two different UV coordinates, then clone that vertex. Each vertex clones at the same index as the UV coordinates. If a UV coordinate is pointing to two different vertices, then clone the UV coordinate. We will make some information redundant and it would be much better than making complete copies for each face.

How do we achieve that? Let's look at the algorithm we designed:

```
verticesFromFaceUvs: function(vertices, uvs, materialIndex) {
  var vertexVectors = [];
  var redundantVertexVectors = [];

  var vertexCovered = [];
  //Copy vertices to a vec3 array
  for (var i = 0; i < vertices.length; i = i + 3) {
    var vector = vec3.fromValues(vertices[i], vertices[i + 1],
vertices[i + 2]);
    vertexVectors.push(vector);
  }
  var count = 0;
  //Iterating over all uv indices for each vertex in a face.
  for (var i = 0; i < this.faceVertexUvs[materialIndex].length;
++i) {
    var face = this.faces[i];
    var textureIndices = this.faceVertexUvs[materialIndex][i];
    var aVertexIndices = ["a", "b", "c"];
    //Iterating over the each vertex of the face.
    for (var j = 0; j < aVertexIndices.length; ++j) {
      var aVertexIndex = aVertexIndices[j];
      //If the new vertex corresponding to texture coordinate
points to same vertex as in redundant array or the corresponding
vertex is not defined in the redundant array.
      if (redundantVertexVectors[textureIndices[aVertexIndex]] ==
face[aVertexIndex] ||
redundantVertexVectors[textureIndices[aVertexIndex]] === undefined)
{
        redundantVertexVectors[textureIndices[aVertexIndex]] =
face[aVertexIndex];
        face[aVertexIndex] = textureIndices[aVertexIndex];
      }
      else {
        // The texture coordinate holds the index of a different
vertex duplicate the uv coordinate
        uvs[materialIndex].push(uvs[materialIndex]
[textureIndices[aVertexIndex]  2]);
        uvs[materialIndex].push(uvs[materialIndex]
[textureIndices[aVertexIndex]  2 + 1]);
        var newIndex = Math.floor(uvs[materialIndex].length / 2) -
1;
        redundantVertexVectors[newIndex] = face[aVertexIndex];
        face[aVertexIndex] = newIndex;
        textureIndices[aVertexIndex] = newIndex;

      }
```

```
    }
  }
  for (var i = 0; i < redundantVertexVectors.length; ++i) {
    var vector = vertexVectors[redundantVertexVectors[i]];
    this.vertices.push(vector[0]);
    this.vertices.push(vector[1]);
    this.vertices.push(vector[2]);
  }
  this.uvs = uvs;
}
```

In the preceding code, we created a new vertex array (`redundantVertexVectors`) by copying the index of the previous array at a new index location as per the UV index.

It is difficult to express the algorithm in words, so let's do a dry run for the custom set of data.

Understanding the algorithm using a dry run

First, we will create a set of data for dry run as follows:

```
vertices = [x0, y0, z0, x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4,
z4]
uvs = [s0, t0, s1, t1, s2, t2, s3, t3, s4, t4, s5, t5, s6, t6]
faces = [[a=0, b=1, c=2], [a=1, b=2, c=3].....]
faceVertexUvs = [[[a=4, b=5, c=6], [a=4, b=6, c=4] …...]]
redundantVertexVectors = []
```

Initially, the `redundantVertexVectors` array is empty. Let's add a loop, `For i=0, aVertexIndex="a":`

```
textureIndices = faceVertexUvs[i], face=faces[i].
textureIndices["a"]~4, face[ aVertexIndex]~0.
```

Take a look at the following code lines:

```
redundantVertexVectors[textureIndices[aVertexIndex]] =
face[aVertexIndex]
redundantVertexVectors[4] = 0
face[aVertexIndex] = textureIndices[aVertexIndex]
face["a"] = 4
```

Now, the new arrays will look like the following:

```
faces=[[a=4,b=1,c=2],[a=1,b=2,c=3].....]
```

```
faceVertexUvs=[[[a=4,b=5,c=6],[a=4,b=6,c=4] …...]]
redundantVertexVectors=[,,,,0]
```

Let's add another loop, `For i=0, aVertexIndex="b"` with the following `if` condition:

```
if(redundantVertexVectors[textureIndices[aVertexIndex]] ==
face[aVertexIndex] ||
redundantVertexVectors[textureIndices[aVertexIndex]] === undefined)
  textureIndices["b"]~5 face[ aVertexIndex]~1,
redundantVertexVectors[5] === undefined

if(undefined == 4 || redundantVertexVectors[5] == -undefined)
  redundantVertexVectors[textureIndices[aVertexIndex]] =
face[aVertexIndex]
  redundantVertexVectors[5] = 1
  face[aVertexIndex] = textureIndices[aVertexIndex]
  face["b"]=5
  faces=[[a=4,b=5,c=2],[a=1,b=2,c=3].....]
  faceVertexUvs=[[[a=4,b=5,c=6],[a=4,b=6,c=4]…...]]
  redundantVertexVectors=[,,,,0,1]
```

Let's add a loop, `For i=0, aVertexIndex="c"`:

```
faces=[[a=4,b=5,c=6],[a=1,b=2,c=3].....]
faceVertexUvs=[[[a=4,b=5,c=6],[a=4,b=6,c=4]…...]]
redundantVertexVectors=[,,,,0,1,2]
```

Let's add a loop, `For i=1, aVertexIndex="a"` with the following `if` condition:

```
if(redundantVertexVectors[textureIndices[aVertexIndex]]==face[aVert
exIndex]||redundantVertexVectors[textureIndices[aVertexIndex]]===un
defined)
```

Here, `redundantVertexVectors[textureIndices[aVertexIndex]]` *`redundantVertexVectors[4]`* `0` and it is defined. As this condition matches, the `if` condition returns `true`. Hence, the redundant vertex array remains the same but the new index of the new vertex is changed in the `faces` array, as shown in the following code:

```
faces=[[a=4,b=5,c=6],[a=4,b=2,c=3].....]
faceVertexUvs=[[[a=4,b=5,c=6],[a=4,b=6,c=4]…...]]
redundantVertexVectors=[,,,,0,1,2]
```

You must have noticed that no vertex was repeated for this case but simply a new index was assigned in accordance to an existing vertex.

Let's add another loop, `For i=1, aVertexIndex="b"` with the following `if` condition:

```
if(redundantVertexVectors[textureIndices[aVertexIndex]]==face[aVert
exIndex]||redundantVertexVectors[textureIndices[aVertexIndex]]===un
defined)
```

Here, `redundantVertexVectors[textureIndices[aVertexIndex]]` *`redundantVertexVectors[6]`* `2` and `faces[1]["b"]` is equal to 2, which again is the same. Hence, the redundant array will not change, only `face["b"]` will point to the new index, as shown in the following code:

```
faces=[[a=4,b=5,c=6],[a=4,b=6,c=3].....]
faceVertexUvs=[[[a=4,b=5,c=6],[a=4,b=6,c=4]…...]]
redundantVertexVectors=[,,,,0,1,2]
```

Let's add a loop, `For i=1, aVertexIndex="c"` with the following `if` condition:

```
if(redundantVertexVectors[textureIndices[aVertexIndex]]==face[aVert
exIndex]||redundantVertexVectors[textureIndices[aVertexIndex]]===un
defined)
```

Here, `redundantVertexVectors[textureIndices[aVertexIndex]]` *`redundantVertexVectors[4]`* `0` and `faces[1]["c"]` is equal to 3 which makes the `if` condition `false`. Hence, in the `else` condition, the following code will be added:

```
uvs[materialIndex].push(uvs[materialIndex]
[textureIndices[aVertexIndex]*2]);
uvs[materialIndex].push(uvs[materialIndex]
[textureIndices[aVertexIndex]*2+1]);
  var newIndex=Math.floor(uvs[materialIndex].length/2)-1;
  redundantVertexVectors[newIndex] = face[aVertexIndex];
  face[aVertexIndex] = newIndex;
  textureIndices[aVertexIndex] = newIndex;
```

Here, `uvs[0].push(uvs[0][4*2~8])` ~ `uvs[0].push(s4)`, `uvs[0].push(uvs[0][4*2+1~9])` ~ `uvs[0].push(t4)`, and `var newIndex = Math.floor(uvs[materialIndex].length/2)-1` ~ 7. The new length of the UV array minus one, `redundantVertexVectors[7] = 4`, `face["c"] = 7`, and `textureIndices["c"]=7`. The resultant arrays are as follows:

```
uvs=[s0,t0, s1,t1, s2,t2, s3,t3, s4,t4, s5,t5, s6,t6, s4,t4]
```

```
faces=[[a=4,b=5,c=6],[a=4,b=6,c=7].....]
faceVertexUvs=[[[a=4,b=5,c=6],[a=4,b=6,c=7]…...]]
redundantVertexVectors=[,,,,0,1,2,4]
```

As you keep iterating through this function, you will realize which condition has the minimum redundancy and you will have the following new arrays:

- A new `redundantVertexVectors` array with indices to the original vector array. We will use this array to create our new vertices array as follows:

```
for(var i=0; i<redundantVertexVectors.length; ++i) {
  var vector=vertexVectors[redundantVertexVectors[i]];
  this.vertices.push(vector[0]);
  this.vertices.push(vector[1]);
  this.vertices.push(vector[2]);
}
```

- A new UV array with some redundant UV coordinates.
- A `faces` array where each face element has indices of the new array for each vertex.

Now, we can create our indices array like we did before, by using the `indicesFromFaces` function of the `Geometry` class with new vertices and face indexes. We will also use the `calculateVertexNormals` function of our `Geometry` class to calculate normals for the new vertices and indices.

Note

The two algorithms that we just discussed to create new vertices and indices arrays are not standard algorithms. We can always write better algorithms to reduce the size of the vertices array.

Rendering objects exported from Blender

If you have reached this far, you have learned the main recipe to fight mutated Mr. Green. Understanding the JSON format and then recreating the vertices and indices arrays is exciting and at the same time, is your biggest weapon to build the most beautiful objects with textures. So let's quickly walk through the code to load JSON objects.

Changes in our JSON parser

Open `primitive/parseJSON.js` in your editor. This file has two major changes. The first change populates the `faceUvs` and `faceVertexUVs` arrays of the `Geometry` class with the JSON data, as shown in the following code snippet:

```
fi = geometry.faces.length;
if ( hasFaceUv ) {
  for ( i = 0; i < nUvLayers; i++ ) {
    uvLayer = data.uvs[ i ];
    uvIndex = faces[ offset ++ ];
    geometry.faceUvs[ i ][ fi ] = uvIndex;
  }
}
if ( hasFaceVertexUv ) {
  for ( i = 0; i < nUvLayers; i++ ) {
    uvLayer = data.uvs[ i ];
    uvs = [];
    var aVertexIndices=["a","b","c","d"];
    for ( j = 0; j < nVertices; j ++ ) {
      uvIndex = faces[ offset ++ ];
      //u = uvLayer[ uvIndex  2 ];
      //v = uvLayer[ uvIndex  2 + 1 ];
      uvs[ aVertexIndices[j] ] = uvIndex;
    }
    geometry.faceVertexUvs[ i ][ fi ] = uvs;
  }
}
```

The second change invokes geometry call functions to create vertices, indices, normals, and UV coordinate arrays, as shown in the following code snippet:

```
geometry.materials = data.materials;
geometry.verticesFromFaceUvs(data.vertices, data.uvs, 0);
geometry.indicesFromFaces();
```

Changes in our Geometry object

Open `primitive/Geometry.js` in your favorite editor. We added a new function, `verticesFromFaceUvs`, to re-create vertices from the UV arrays as discussed previously in the chapter.

Loading a textured object

Let's now load an object with texture mapping exported from Blender. Open `04-Loading-Model-Textures.html` in your editor. The code of the shaders remains the same as the previous code snippets (`04-SquareWithTexture.html`) of this chapter. A new attribute, `vec2 aTextureCoord`, to hold texture coordinates and a new varying, `vTextureCoord`, is added to the vertex shader to pass the coordinate to the fragment shader. In the fragment shader, the code was changed to calculate the new fragment color value. It is used in the texture access function, `texture2D`, to extract the color value from the uniform, `uSampler`, as shown in the following code snippet:

```
gl_FragColor = vec4(iColor, 1.0)*texture2D(uSampler,
vec2(vTextureCoord.s, vTextureCoord.t));
```

In the main control code for the textured object, the load model function takes the URL of the JSON model. It first invokes the `parseJSON` function which returns the geometry object pre-populated with the vertex data information. Then, it retrieves the texture URL from the geometry object, `geometry.materials[materialIndex].mapDiffuse`, and it invokes `initTexture` with the texture URL. The `initTexture` function loads the texture and sets global parameters such as `gl.UNPACK_FLIP_Y_WEBGL` to flip the texture to match the texture coordinates of WebGL in the `handleTextureLoaded` function. Then, the code loads the texture into the GPU memory and sets texture parameters, as shown in the following code snippet:

```
function initTextures(imageMap) {
  texture = gl.createTexture();
  image = new Image();
  image.onload = function() { handleTextureLoaded(image, texture);
  }
  image.src = path+imageMap;
}
function handleTextureLoaded(img, texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, img);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);
  gl.bindTexture(gl.TEXTURE_2D, null);
```

```
  initScene();
}
function loadModel(url){
  rotateX=0.0;
  rotateY=0.0;
  $.getJSON(path+url, function(data) {
    geometry=parseJSON(data);
    if(geometry.materials.length>0) {
      if(!(geometry.materials[0].colorDiffuse===undefined))
        diffuseColor=geometry.materials[0].colorDiffuse;
      if(!(geometry.materials[0].colorAmbient===undefined))
        ambientColor=geometry.materials[0].colorAmbient;
      if(!(geometry.materials[0].colorSpecular===undefined))
        specularColor=geometry.materials[0].colorSpecular;
    }
    initTextures(geometry.materials[materialIndex].mapDiffuse);
  });
}
```

The `initBuffers` function creates vertex buffers (`gl.ARRAY_BUFFER`) for vertices, normals, and texture coordinates. It initializes the buffers with geometry variables (vertices, normals, and UVs). Then, it creates the index buffer (`gl.ELEMENT_ARRAY_BUFFER`) and loads `geometry.indices` into it, as shown in the following code snippet:

```
function initBuffers() {
  vertexPositionBuffer = gl.createBuffer();
  indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(geometry.vertices), gl.STATIC_DRAW);
  vertexPositionBuffer.itemSize = 3;
  vertexPositionBuffer.numItems = geometry.vertices.length/3;
  vertexNormalBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexNormalBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(geometry.normals), gl.STATIC_DRAW);
  vertexNormalBuffer.itemSize = 3;
  vertexNormalBuffer.numItems = 24;
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
Uint16Array(geometry.indices), gl.STATIC_DRAW);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
  verticesTextureBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, verticesTextureBuffer);
```

```
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(geometry.uvs[materialIndex]), gl.STATIC_DRAW);
}
```

The `drawScene` function remains the same, except in the `drawElements` call where we pass `geometry.indices` which was initialized in the `parseJSON` function, as shown in the following code snippet:

```
function drawScene() {
  ...
  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.uniform1i(gl.getUniformLocation(shaderProgram, "uSampler"),
0);
  setMatrixUniforms();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
  gl.drawElements(gl.TRIANGLES, geometry.indices.length,
gl.UNSIGNED_SHORT, 0);
}
```

When we load the image in our browser, we see that the texture is pixelated. We had briefly discussed filter modes earlier and in the next section, we will dive deep into the different filter modes.

Understanding mipmapping

First, let's understand what texture filtering and mipmapping are. Take a close look at the following images. The image on the left shows the image of a monkey model with aliasing effects (look at the eyes of the monkey). On the right, we have an image of the same model with no such effects. When the object moves closer to the camera or moves further away from it, the texture associated with that object appears blocky or pixelated. Let's first understand why this happens. In our shader, we pass the image coordinates of the image. Now the shader interpolates texel values (texture pixels) and decides which pixel of the image to pick to be shaded on the screen. When the object is close to the viewer, each texel might become larger than the screen pixel. When the object is very far from the viewer, the screen pixel might become larger than the texel. Now, when it comes to selecting the texel, the rendering engine needs to decide which algorithm to follow to select the texel. The algorithms have to be decided by us, the programmers. Each algorithm has its pros and cons. Some are very accurate but processor-intensive, and some are inaccurate but fast. In this section, we will learn about each of them, and then we can decide which one to use for rendering a particular object, depending on their importance in the game. The best part is you get to apply various filtering modes for different textures.

Implementing mipmapping

We provide optimized collections of images that accompany a main texture to increase the rendering speed and reduce aliasing artifacts. These optimized images are collectively called a mipmap set. If the texture has a basic size of 1024 × 1024 pixels, then the associated mipmap set may contain a series of nine images, and each takes up one-fourth of the total area of the previous image: 512 × 512 pixels, 128 × 128, 64 × 64, 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2, and 1 × 1. These images are generally computed by the rendering engine. The renderer will switch to a suitable mipmap image when the texture is viewed from a distance or is of a small size. Artifacts like blurring, shimmering, and blocking are minimized considerably when images of different sizes are used. Also the rendering speed is considerably enhanced as the engine uses smaller texture sizes for objects at far-off distances.

The function used to automatically generate a mipmap set in the WebGL API is `gl.generateMipmap(gl.TEXTURE_2D)`.

It will generate mipmap for the current texture buffer object. WebGL can generate the mipmap images automatically, but in this case, we manually upload a series of images in order to have more control. In the function, `gl.texImage2D`, the second parameter is `level`. This `level` parameter refers to the level of detail. So, we can upload multiple images for multiple mipmap levels and assign them to the same texture buffer object, as shown in the following code snippet:

```
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
img);
gl.texImage2D(gl.TEXTURE_2D, 1, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
img256);
gl.texImage2D(gl.TEXTURE_2D, 2, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
img128);
gl.texImage2D(gl.TEXTURE_2D, 3, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
img64);
gl.texImage2D(gl.TEXTURE_2D, 4, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
img32);
```

In the preceding code, the current buffer object is `texture`. We upload multiple mipmap images (`img`, ..., `img32`) for the same buffer object with different **level of detail** (**LOD**) values.

Understanding the filtering methods

Now let's understand how mipmap sets are used in filtering methods available in the WebGL API.

Nearest-neighbor interpolation

Nearest-neighbor interpolation is the fastest and simplest filtering method. It simply uses the color of the texel closest to the pixel center for the pixel color. While being fast, it results in a large number of artifacts. Its usage is shown as follows:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

Linear interpolation

In the linear interpolation method, the four nearest texels to the pixel center are sampled and their colors are combined by the weighted average of their distance. This removes the blockiness seen during magnification. There is a smooth gradient of color change from one texel to the next. It is also called bilinear filtering. Its usage is shown as follows:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

Nearest-neighbor with mipmapping

In the nearest-neighbor with mipmapping algorithm, the nearest mipmap level is first chosen, then the nearest texel center is used to get the pixel color. This reduces the aliasing and shimmering significantly but does not help with blockiness. Its usage is shown as follows:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_NEAREST);
```

Bilinear filtering with mipmapping

In bilinear filtering, the four nearest texels to the pixel center are sampled at the closest mipmap level. The colors of the texels are combined by weighted average of their distance. This removes the blockiness seen during magnification. So, instead of an abrupt color change, there is now a smooth gradient of color change from one texel to the next. Its usage is shown as follows:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR_MIPMAP_NEAREST);
```

Trilinear filtering

Trilinear filtering is a solution to a common problem seen in bilinearly filtered mipmapped images. A very noticeable change is observed in quality when the WebGL renderer switches from one mipmap level to the next. Trilinear filtering first does a texture lookup and then applies either bilinear or nearest filtering on the two closest mipmap levels. It then linearly interpolates the resulting values. This results in a smooth degradation of the texture quality as the distance from the viewer increases rather than a series of sudden drops.

When two mipmaps are selected and we want to apply the nearest-neighbor algorithm, we use the following code snippet:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_LINEAR);
```

When two mipmaps are selected and we want to apply the bilinear algorithm, we use the following code:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR_MIPMAP_LINEAR);
```

Note

Mipmapping and its corresponding filtering modes can only be applied to POT (power of two) images, that is, images that have a height and width in the power of two (64 × 64, 256 × 128, and so on). We can apply only the nearest and linear filtering modes to non-POT images.

Applying filtering modes

To apply filtering modes to our texture, we will add three dropdowns:

- **Distance**: This dropdown will help us move the object closer or farther from the viewer
- **Magnification Filtering**: This dropdown holds the magnification filtering modes
- **Minification Filtering**: This dropdown holds the minification filtering modes

Open the `04-ApplyingFilteringModes.html` file in your editor. We have changed the code to add the dropdowns, as shown in the following code snippet:

```
<div>
  <label>Minification Filter</label>
  <select id = "minificationFilter">
    <option value = "NEAREST">NEAREST</option>
    <option value = "LINEAR">LINEAR</option>
    <option value =
"NEAREST_MIPMAP_NEAREST">NEAREST_MIPMAP_NEAREST</option>
    <option value =
"LINEAR_MIPMAP_NEAREST">LINEAR_MIPMAP_NEAREST</option>
    <option value =
"NEAREST_MIPMAP_LINEAR">NEAREST_MIPMAP_LINEAR</option>
    <option value =
"LINEAR_MIPMAP_LINEAR">LINEAR_MIPMAP_LINEAR</option>
  </select>
  <label>Maginification Filter</label>
  <select id="magnificationFilter">
    <option value="NEAREST">NEAREST</option>
    <option value="LINEAR">LINEAR</option>
  </select>
</div>
<div>
  <label>Select Distance</label>
  <select id="distanceList">
    <option value="-1">1</option>
  ...
    <option value="-12">12</option>
  </select>
</div>
```

We have added the `change` function to handle change events in the `start`

function using jQuery selectors, as shown in the following code snippets:

```
function start() {
  ...
  $("#minificationFilter").change(changeValues);
  $("#magnificationFilter").change(changeValues);
  $("#distanceList").change(changeValues);
  ...
}
```

We have added a function, `changeValues`, to set the texture parameters' filtering modes and to set the value of the global variable, `distance`. We retrieve the values of the dropdown using jQuery selectors. The `texture` buffer is made the current buffer using the API call, `bindTexture`, for subsequent `gl.texParameteri` calls. The switch case statement sets the `gl.TEXTURE_MIN_FILTER` and `gl.TEXTURE_MAG_FILTER` parameters for the current texture. Then, we invoke the `drawScene` function to see the effect of these parameters, as shown in the following code snippet:

```
function changeValues() {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  var minification=$("#minificationFilter option:selected").val();
  var magnification=$("#magnificationFilter
option:selected").val();
  distance=parseInt($("#distanceList  option:selected").val());
  switch(minification) {
    case "NEAREST":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST);
      break;
    case "LINEAR":
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR);
      break;
    case "NEAREST_MIPMAP_NEAREST":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_NEAREST);
      break;
    case "LINEAR_MIPMAP_NEAREST":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR_MIPMAP_NEAREST);
      break;
    case "NEAREST_MIPMAP_LINEAR":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_LINEAR);
      break:
```

```
    case "LINEAR_MIPMAP_LINEAR":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR_MIPMAP_LINEAR);
      break;
  }
  switch(magnification) {
    case "NEAREST":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
      break;
    case "LINEAR":
      gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
gl.LINEAR);
      break;
  }
  gl.bindTexture(gl.TEXTURE_2D, null);
  drawScene();
}
```

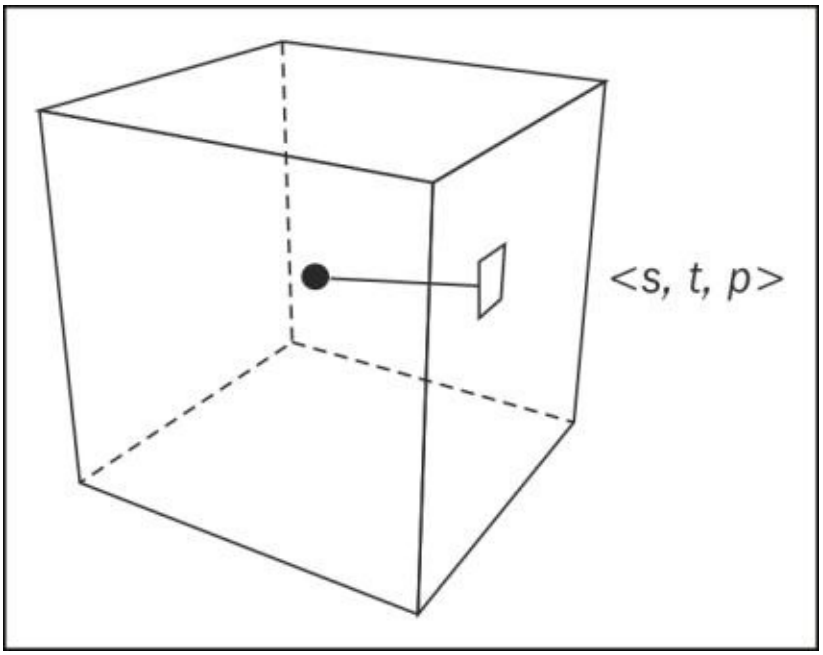
Change the `drawScene` function to translate `mvMatrix` by the `distance` variable set in the `changeValues` function. This translation is like moving the camera closer or farther from the model. The required change is shown as follows:

```
mat4.translate(mvMatrix, mvMatrix, [0.0, 0.0, distance]);
```

Understanding cubemaps and multi-texturing

So far, we have used only 2D textures in our model. However, at the beginning of the chapter, we also mentioned another kind of texture called cubemaps. Cubemaps are often used in games to approximate an environmental reflection on the surface of a model. Let's say you have a shiny mirror in your game and you want to show the reflection of the left wall, or when the mirror rotates, you want to show the reflection of the right wall. The biggest application of cubemaps in gaming is skylight illumination and skyboxes.

A cube texture map consists of six two-dimensional images that correspond to the faces of a cube. The *s*, *t*, and *p* coordinates represent a direction vector emanating from the center of the cube that points toward the texel to be sampled, as shown in the following figure:



For cubemap sampling, first we need to select a face, and then use 2D coordinates to sample a color value from the selected image. Which face to sample is determined by the sign of the coordinate with the largest absolute value. The other two coordinates are divided by the largest coordinate and

remapped to the range 0 to 1 using formulas. Well, we do not need to do any of the calculations or even need to know the formulas. The GPU does all that for us. We only need to specify the 2D textures of the six sides of the cubemap. We specify each face of the cubemap with positive-Z, positve-Y, positiveX, negativeX, negative-Y, and negative-Z directions. The `texImage2D` function is called as follows:

```
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_X, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);
```

Cubemap coordinates

We do not provide the cubemap coordinates to the shader. In fact, the cubemap colors are sampled using the vertex normals using the following function:

```
textureCube(uCubeSampler, transformedNormal);
```

The `textureCube` function samples the color from the location where the normal from the object surface intersects the cubemap. The `uCubeSampler` uniform references the cubemap texture.

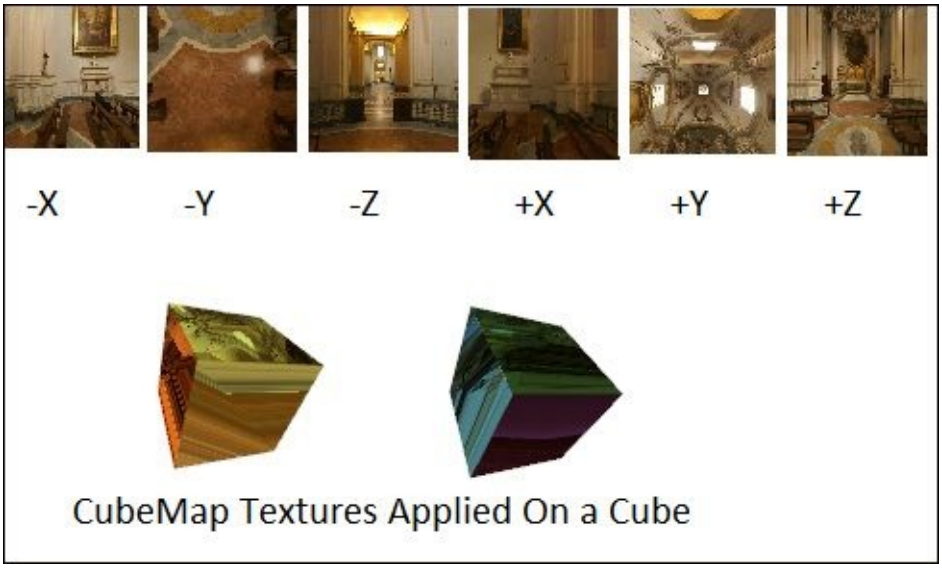
Multi-texturing

So far, we have been rendering using a single texture. However, there are times where we may want to have multiple textures that contribute to a fragment to create more complex effects. For such cases, we use WebGL's ability to access multiple textures in a single draw call. This is known as multi-texturing. In this chapter, we earlier read that WebGL can refer 32 active textures and each texture is represented by `TEXTURE0` through `TEXTURE32`. We can assign these textures to sampler uniforms in our fragment shader and use them simultaneously to color a fragment.

Loading cubemaps

In our code, we will show how to use cubemaps and discuss multi-texturing. So far, we have used a single texture in our code examples. However, in this code, we will load two textures. The first will be our 2D texture and the second will be our cubemap applied to the same object.

The following figure shows the different images used for the cubemaps and the corresponding object on which they are applied:



Understanding the shader code

Open `04-ApplyingCubeMaps.html` in your favorite editor. The vertex shader remains the same as in our previous examples of this chapter. The changes in the fragment shader are as follows:

```
uniform samplerCube uCubeSampler;
```

A new uniform variable of the `samplerCube` type is defined in the code. The value of `gl_FragColor` is deduced as follows:

```
gl_FragColor = vec4(iColor, 1.0)  texture2D(uSampler,
vec2(vTextureCoord.s, vTextureCoord.t))  textureCube(uCubeSampler,
transformedNormal);
```

Now, `gl_fragColor` has a new component, `textureCube(uCubeSampler, transformedNormal)`. The texture calls the `textureCube` function, which samples the color values from the uniform, `uCubeSampler`. The `transformedNormal` parameter is passed as a varying from the vertex shader. We were already using this for lighting calculations.

We have also added two functions: `loadCubeMap` and `loadFace`. These functions initialize our cubemap texture. We first create a memory reference in GPU for the cubemap texture using the `bindTexture` call. We now pass `gl.TEXTURE_CUBE_MAP` as the first parameter instead of `gl.TEXTURE_2D`. We use the nearest-neighbor interpolation filtering mode for minification and the bilinear interpolation for magnification. Then, we load 2D textures for each face of the cubemap. Each face is loaded in the GPU and is associated with the faces of the cubemap by specifying the target as `gl.TEXTURE_CUBE_MAP_POSITIVE_X` or `gl.TEXTURE_CUBE_MAP_POSITIVE_Y` in the `gl.texImage2D` API call. After each texture is loaded, we invoke the `drawScene` function to see the effect, as shown in the following code snippet:

```
function loadFace(target, url) {
  var image = new Image();
  image.onload = function() {
    gl.bindTexture(gl.TEXTURE_CUBE_MAP, cubeTex);
    gl.texImage2D(target, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
image);
    gl.bindTexture(gl.TEXTURE_CUBE_MAP, null);
    drawScene();
  }
  image.src = url;
};
function loadCubeMap() {
  cubeTex = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_CUBE_MAP, cubeTex);
  gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MAG_FILTER,
gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MIN_FILTER,
gl.LINEAR);
  loadFace(gl.TEXTURE_CUBE_MAP_POSITIVE_X,
'cubemap/positive_x.jpg');
  loadFace(gl.TEXTURE_CUBE_MAP_NEGATIVE_X,
'cubemap/negative_x.jpg');
  loadFace(gl.TEXTURE_CUBE_MAP_POSITIVE_Y,
'cubemap/positive_y.jpg');
  loadFace(gl.TEXTURE_CUBE_MAP_NEGATIVE_Y,
'cubemap/negative_y.jpg');
```

```
  loadFace(gl.TEXTURE_CUBE_MAP_POSITIVE_Z,
'cubemap/positive_z.jpg');
  loadFace(gl.TEXTURE_CUBE_MAP_NEGATIVE_Z,
'cubemap/negative_z.jpg');
}
```

In the `drawScene` function, we initialize our two textures as follows:

```
function drawScene() {
  ...
  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.uniform1i(gl.getUniformLocation(shaderProgram, "uSampler"),
0);
  gl.activeTexture(gl.TEXTURE1);
  gl.bindTexture(gl.TEXTURE_CUBE_MAP, cubeTex);
  gl.uniform1i(gl.getUniformLocation(shaderProgram,
"uCubeSampler"), 1);
  ...
}
```

In the preceding code, we activate our first texture, and then we associate our current 2D texture by using the parameter `0` in the `gl.uniform1i(..., 0)` call. Similarly, we activate our second texture `gl.activeTexture(gl.TEXTURE1)` and then associate our cubemap texture as `1` in the `gl.uniform1i(..., 1)` call.

Notice the effect of the following cubemap texture (hall in Rome) and 2D texture applied on the cube model as shown in the following figure:

Summary

In gaming applications, textures are the key factors that affect the performance as well as the look of the game. The size of the texture and intelligent use of concepts, such as mipmapping, will decide how your game would perform. Also, we learned how to map texture coordinates to vertex coordinates, which is a very important concept to control the overall memory footprint of your objects in the GPU. If you plan to use open source WebGL libraries to create your games, these concepts will come in handy as you might have to tweak them to get the last bit of performance from your game.

In the next chapter, we will learn about the different types of camera used in 3D games. We will also write a new `Camera` class to view our world in 5000 AD with different perspectives.

Chapter 5. Camera and User Interaction

It is impossible to imagine a 3D game without a camera. WebGL does not provide a camera class. We have learned that WebGL is a low-level API, but it gives us a rendering API to help us write one of our own implementations. This chapter is focused on evolving our own camera class for our game scene. We will also empower our users to view the game scene from different angles and positions by adding mouse and keyboard interactivity. We will also implement different types of cameras used in gaming. The topics we will cover are as follows:

- ModelView transformations
- Perspective transformations
- The basic camera
- The free camera
- Controlling the camera with the keyboard and mouse
- The orbit camera

Understanding ModelView transformations

In all our previous chapters, we used model transformations. We used `glMatrix` (http://glmatrix.net/) functions to calculate translation and rotation. Let's look at what we are talking about here:

```
// Create an identity matrix
var mvMatrix=mat4.create();

// Translate the matrix by objects location and replace the matrix
with the result(mat4.translate(out,in,vec3))
  mat4.translate(mvMatrix, mvMatrix,
stage.stageObjects[i].location);
// Rotate the matrix by objects rotation X and replace the matrix
with the result(mat4.rotateX(out,in,vec3))
  mat4.rotateX(mvMatrix, mvMatrix,
stage.stageObjects[i].rotationX);
  mat4.rotateY(mvMatrix, mvMatrix,
stage.stageObjects[i].rotationY);
  mat4.rotateZ(mvMatrix, mvMatrix,
stage.stageObjects[i].rotationZ);
```

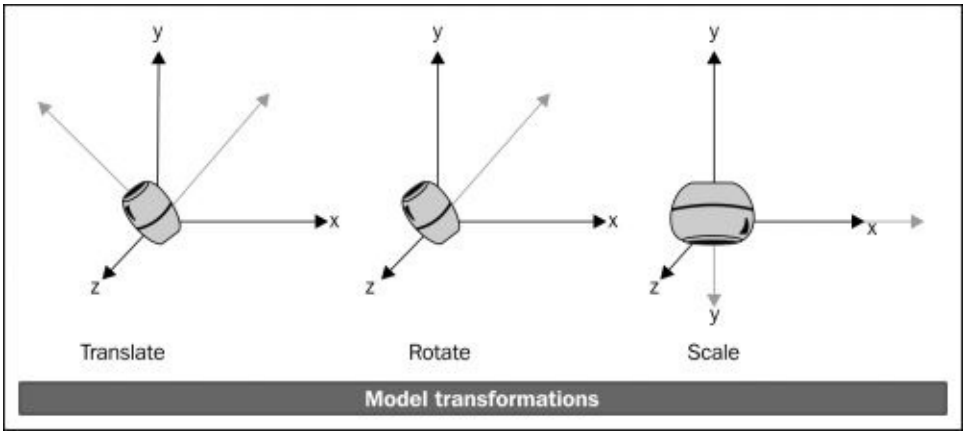
In the preceding code, we created a ModelView matrix, `mvMatrix`, which stores the translation and rotation of the object. We passed this matrix to our fragment shader using our own function, `setMatrixUniforms`. When we invoked `drawElements`, each vertex of the object was transformed in the vertex shader because of the multiplication of each vertex location with the `mvMatrix`, as shown in the following code:

```
vec4 vertexPos4 = mvMatrix * vec4(aVertexPosition, 1.0);
```

So why is this matrix called a ModelView matrix when it stores only the model transformations? Well, this is because until now, we have not considered the view matrix. We have only been viewing our objects from one angle and location. The view matrix and object matrix are combined to render our scene ($v'=Mv$). Let's define these two transformations.

Applying the model transformation

We use the modeling transformation to position and orient the model. For example, we can translate, rotate, or scale the model or perform a combination of these operations. These three operations are depicted in the following figure:



Understanding the view transformation

The viewing transformation is analogous to the positioning and aiming of a camera. In our code examples, the current matrix is set to the identity matrix using `mat4.identity(mvMatrix)`. This step is necessary as most of the transformation commands multiply the current matrix by the specified matrix and then set the result to be the current matrix. So essentially, we can say that until now we were setting our view matrix to the identity matrix. So, let's dive deep into the view matrix and its calculations.

The view matrix is the most misunderstood concept in 3D game programming. The camera transformation matrix defines the position and orientation of the camera in the world space. However, we use the view matrix to transform the model's vertices from world space to view space. The camera matrix and view matrix are not the same.

To understand it, let's define the two terms separately:

- **Camera transformation matrix**: This is exactly like the model matrix. We treat our camera like an object. It has a location and can be rotated in a scene. The final transformation matrix is our camera matrix.
- **View matrix**: This matrix is used to transform the vertices of a model from world space to view space. This matrix is the inverse of the camera transformation matrix. In the real world, you move the camera but in 3D rendering, we say that the camera does not move but the world moves in the opposite direction and orientation. Hence, to transform the model to view space, we use the inverse of the camera matrix.

Understanding the camera matrix

The camera transformation matrix is exactly the same as the model transformation matrix, except that we do not use the scale component to calculate it. Let's reiterate the components used to calculate a model transformation matrix. A model matrix is calculated using the translation, rotation, and scale functions. In the camera matrix, we generally do not have a scale functions.

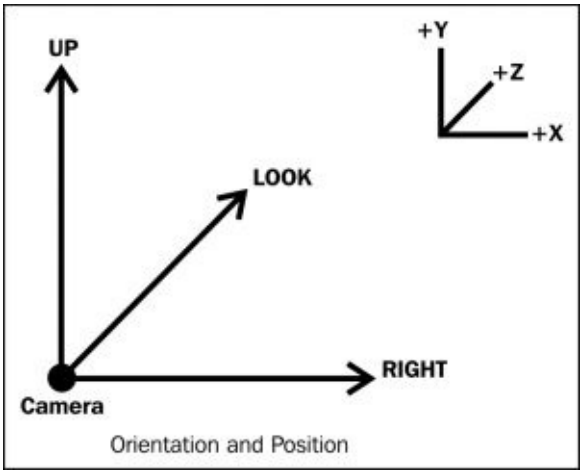
Hence, if *R* represents the rotation matrix of the camera and *T* represents the translation matrix in world space, then the final transformation matrix *M* can be computed by multiplying the two matrices, *M = T*R*. To get the position of the camera in world-space (also called the eye position), from the matrix *M*, you simply take the fourth row of the resulting 4 × 4 matrix, M = (M_{41}, M_{42}, M_{43}, M_{44}).

Comprehending the components of a camera matrix

A camera matrix is represented by four orthogonal unit vectors. These vectors define the orientation and position of the camera in the scene. The **look** vector represents the direction the camera is pointing to. The **up** vector denotes the upward direction. The **right** vector points to the right of the camera, and finally the **position** element says where the camera is positioned in the world. The breakdown of the final view matrix is as follows:

Right.x	Up.x	Look.x	0.0f
Right.y	Up.y	Look.y	0.0f
Right.z	Up.z	Look.z	0.0f
Positon.x	Positon.y	Positon.z	1.0f

The following figure shows the three orthogonal vectors (**Up**, **Look**, and **Right**). The vectors are orthogonal if their dot product is 0 (perpendicular to each other).



Orientation and Position

Once we set our four orthogonal vectors, we can calculate our camera matrix, as follows:

```
var cameraMatrix= -mat4.create();
  mat4.multiplyVec4(cameraMatrix, [1, 0, 0, 0], this.right);
  mat4.multiplyVec4(cameraMatrix, [0, 1, 0, 0], this.up);
  mat4.multiplyVec4(cameraMatrix, [0, 0, 1, 0], this.normal);
  mat4.multiplyVec4(cameraMatrix, [0, 0, 0, 1], this.position);
```

Converting between the camera matrix and view matrix

We calculate the view matrix by computing the inverse of the camera transformation matrix, $V=M^{-1}$, as shown in the following code snippet:

```
var viewMatrix=mat4.create();
mat4.invert(viewMatrix, cameraMatrix);
```

If we have the view matrix, we can calculate the camera transformation matrix by computing the inverse view matrix. Using the camera matrix, we can derive the position and orientation of the camera, $M=V^{-1}$, as shown in the following code snippet:

```
mat4.invert(cameraMatrix, viewMatrix);
```

Now let's look at a simple function that we evolved to compute the view matrix:

```
var m = mat4.create();
mat4.inverse(m, cameraMatrix);
return m;
```

Note

We do not recommend that you use the preceding code to create the view matrix. However, the code is important as we will use the explanation to calculate the model matrix of stage objects. We have used a handy function in the `glMatrix` library, the `lookAt` function to calculate the view matrix. We will use this function in our game.

Using the lookAt function

In this section, we will understand how `glMatrix` calculates the view matrix from the orthogonal vectors. Take a look at the following code:

```
var matView=mat4.create();
var lookAtPosition=vec3.create();
vec3.add(lookAtPosition, this.pos, this.dir);
mat4.lookAt(matView, this.pos, lookAtPosition, this.up);
```

The `lookAt` function takes three parameters and then computes our `matView` matrix (view matrix). The first parameter is `position` that represents the position of the camera, the second parameter is `lookAtPosition` that denotes the point in the world scene that our camera is looking at, and the third parameter is the `up` vector. For a generic camera, `lookAtPosition` is calculated by adding the position, `this.pos`, and the look vector of the camera, `this.dir` in our case.

The following algorithm explains how the `lookAt` function computes the view matrix:

```
Vector3 zaxis = normal( lookAtPosition – this.pos); // The "look-
at" vector.
Vector3 xaxis = normal(cross(this.up, zaxis)); // The "right"
vector.
Vector3 yaxis = cross(zaxis, xaxis); // The "up" vector.

// Create a 4 x 4 orientation matrix from the right, up, and at
vectors
    Matrix4 orientation = {
        xaxis.x, yaxis.x, zaxis.x, 0,
        xaxis.y, yaxis.y, zaxis.y, 0,
        xaxis.z, yaxis.z, zaxis.z, 0,
          0,       0,       0,     1
    };
    // Create a 4 x 4 translation matrix by negating the eye
position.
    Matrix4 translation = {
          1,      0,      0,     0,
          0,      1,      0,     0,
          0,      0,      1,     0,
        -pos.x, -pos.y, -pos.z,  1
    };
    // Combine the orientation and translation to compute the view
matrix
    return ( translation * orientation );
```

```
return ( translation * orientation );
```

The function first calculates our three vectors (look, up, and right). If you look at the figure above the direction, the look vector represents the *z* axis of the camera and the up vector represents the *y* axis. First, we calculate the *z* axis (the look or direction vector) by subtracting the position of the camera from `lookAtPosition`. As we mentioned earlier, each vector is orthogonal, so if we have two vectors, we can calculate the third. In order to calculate the *x* axis, we find the cross product of the look and the up vectors (`yaxis = cross(zaxis, xaxis)`). Now we readjust our *y* axis with respect to our computed *z* and *x* axes.

The function then creates the orientation matrix and translation matrix using the position and orthogonal vectors. Then, we calculate the view matrix by multiplying the translation with the orientation matrix.

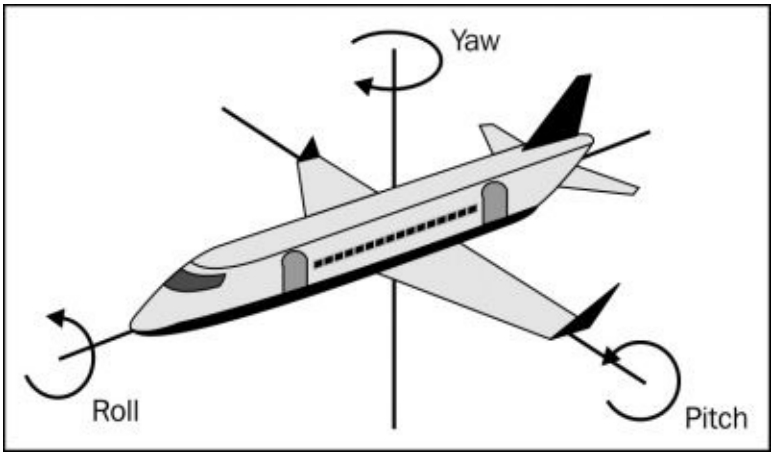
Note

The preceding algorithm is already implemented in the `lookAt` function in `glMatrix`. We have presented this algorithm to explain the logic to calculate the vectors (direction, up, and right) from `lookAtPosition`, the camera position, and the up vector. This logic will be instrumental in understanding the implementation of orbit cameras.

Understanding the camera rotation

To rotate the camera, we need to rotate the up, look, or the right vector. To do this, we create a matrix describing our rotation and transform the relevant vectors using it.

We will call the rotation around the look vector (or *z* axis) **Roll**. The rotation around the up vector (or *y* axis) is called **Yaw**. The rotation around the right vector (*x* axis) is called **Pitch**. The following is a picture of an aircraft showing **Roll**, **Pitch**, and **Yaw** in terms of the aircraft's local axes:



Most developers use the airplane figure to represent pitch, yaw, and roll, as the terms are mostly used in flight dynamics.

Using quaternions

We represent our rotations in an orthogonal matrix, whose rows or columns are orthogonal vectors. We can also represent our 3D rotations using Euler angles. Euler angles help us define the rotation in the three-dimensional world space using only three numbers (yaw, roll, and pitch). However, doing the mathematics using Euler angles results in a problem called gimbal lock when animating objects. So, we generally use other formats to store object rotations. To understand gimbal lock, let's look at the preceding image. If the aircraft pitches up 90 degrees, the aircraft and platform's **Yaw** axis becomes parallel to the **Roll** axis, and changes about the yaw can no longer be compensated for. Well, it is difficult to explain gimbal lock. We would surely observe one in our game development career if we stick to the matrices to represent rotations. For instance, when you try to animate the rotation of a player's arm, you will observe that while incrementing angles along one single plane, the rotation would stop or start at a different angle at a particular instance. However for now, we assume that there are gimbal locks. If you want to know more, refer to http://en.wikipedia.org/wiki/Gimbal_lock.

When using matrices for rotations, we need some other formats to represent rotations in our game. We prefer to use a quaternion to avoid a gimbal lock.

A quaternion represents two things. It has the *x*, *y*, and *z* components, which represents the axis about which a rotation will occur. It also has a *w* component, which represents the amount of rotation that will occur about its axis. In short, a quaternion is a vector and a float.

A quaternion is technically four numbers, three of which have an imaginary component (*i* is defined as *sqrt (-1)*). Well, with quaternions, *i* equals *j* equals *k* equals *sqrt (-1)*. The quaternion itself is defined as $q = w + xi + yj + zk$ where *w*, *x*, *y*, and *z* are all real numbers. We will store a quaternion in a class with four member variables: float *w*, *x*, *y*, and *z*.

Well, we are saved from all the math involved as our `glMatrix` library gives us all the functions required for our computations. Let's look at a few functions:

```
quat.rotateX(out, a, rad) // Rotates a quaternion by the given
angle about the X axis
quat.rotateY(out, a, rad) // Rotates a quaternion by the given
angle about the Y axis
quat.rotateZ(out, a, rad) // Rotates a quaternion by the given
```

```
quat.rotateZ(out, a, rad) // Rotates a quaternion by the given
angle about the Z axis
quat.setAxisAngle(out, axis, rad) // Sets a quat from the given
angle and rotation axis, then returns it.
quat.setAxisAngle(out, [1,0,0], rad) // Sets a quat from the given
angle and X axis, then returns it.
vec3.transformQuat(out, a, q) // Transforms the vec3 with a quat
mat3.fromQuat(out, q) // Calculates a 3 x 3 matrix from the given
quaternion
quat.fromMat3(out, m) // Creates a quaternion from the given 3 x 3
rotation matrix.
```

So basically, if you look at the preceding functions, we can get a matrix from a quaternion and create a quaternion from a rotation matrix. We can rotate our vector using a quaternion. We can create a quaternion for rotation around an axis using the `setAxisAngle` function of our `quat` class. We will soon be putting these functions to action.

Understanding perspective transformations

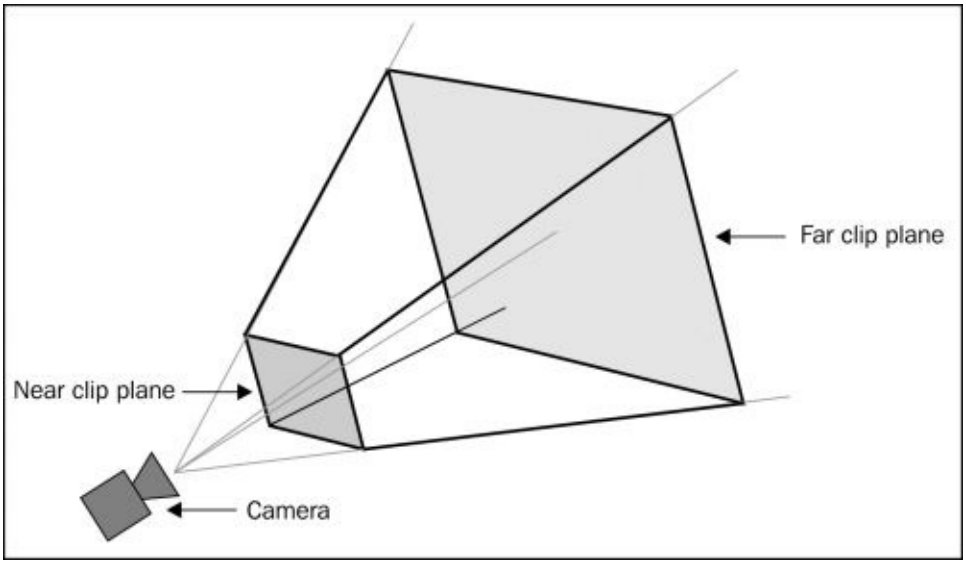
Although we touched upon perspective transformations in Chapter 1, *Getting Started with WebGL Game Development*, and have been using our perspective matrix throughout our code, we would like to discuss it in depth here.

Nearly all 3D games use a perspective projection to render their scene. Like the real world, this simulates the application of a perspective to objects rendered within the game so that objects that are further away appear smaller than objects that are nearer.

In addition to this obvious size effect, more subtle effects of perspective are picked up intuitively by us, and we will add a substantial feeling of depth to the rendered scene. The sides of a cube would seem to narrow slightly as the distance from the viewer increases, thereby allowing us to automatically determine the exact position in which the cube is situated.

Understanding the viewing frustum

The viewing volume of a camera takes the shape of a frustum which is a rectangular cone with its tip cut off, as shown in the following figure:

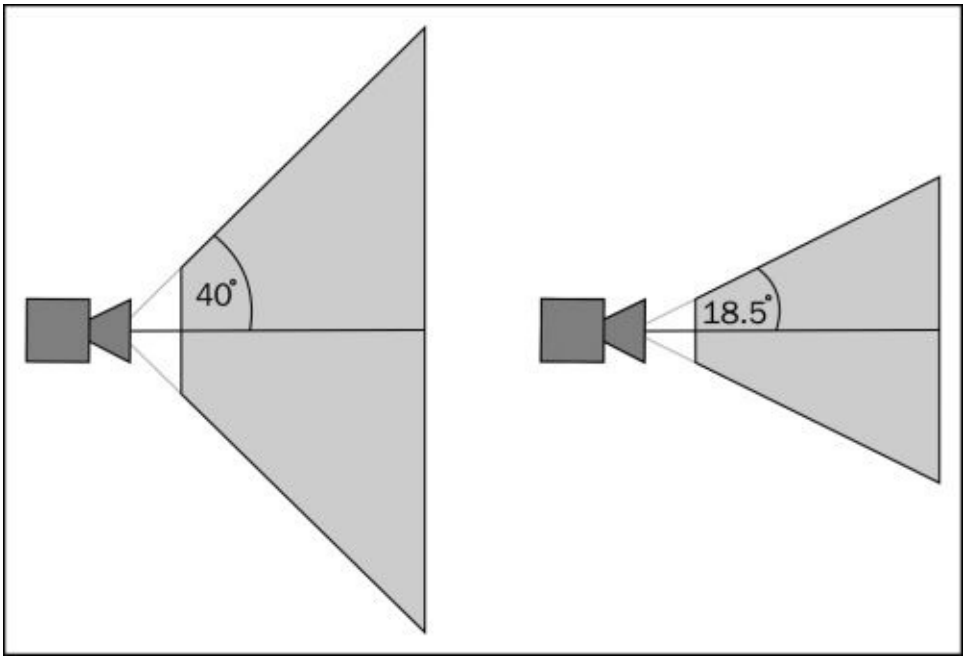


Objects that fall inside the volume of the frustum are visible through the camera. Objects that fall outside the view volume are hidden.

The near and far clip planes decide the visibility of the object. Objects nearer to the camera than the near clip plane and the objects further than the far clip plane are excluded from rendering. In addition to the clip planes, the frustum is defined by the viewing angle and the aspect ratio. The viewing angle defines the angle, in degrees, between the camera and the *y* axis.

Changing the angle will cause the overall shape of the frustum to expand or compress, thereby causing apparent changes in the size of objects.

The following figure shows two viewing frustums from the side view. The first has a viewing angle of 40 degrees and the other has an angle of 18.5 degrees. The distance of the near and far clip planes is the same in both cases.



A distant object will appear larger but will disappear if moved away from the *x* axis in the smaller field of view (FOV). The objects will disappear quickly if moved away from the camera but will be visible when moved farther away from the *x* axis in the larger field of view.

The value we would specify for the viewing angle will vary from one game to the next. An angle within the range of 45 and 60 degrees are usually safe to use.

The aspect ratio is calculated by dividing the frustum's width by its height. The aspect ratio is used to calculate the viewing angle on the *x* axis. The aspect ratio, the viewing angle, and the distances of the near and far planes are used to calculate the projection matrix. The projection matrix is used to depict 3D objects on a 2D screen.

Defining the view frustum

Now, let's again look at the `glMatrix` function we have used throughout our code:

```
mat4.perspective(this.projMatrix, degToRadian(this.fieldOfView),
aspectRatio, this.nearClippingPlane, this.farClippingPlane)
```

The four parameters that we discussed are passed to the preceding function. The field of view, the aspect ratio (width/height of our canvas), and the distances of the near and far clipping planes are passed to create our perspective transformation matrix.

Using the basic camera

Let's now write our class for the basic camera. What have we learned so far? There are two transformation matrices: the view matrix and the projection matrix. To calculate the view matrix, we need four parameters: up, direction, left, and position. These parameters define the camera's orientation and position in space.

To compute the projection matrix, we need four parameters as well: field of view (radians), aspect ratio of the 2D screen, and the near and far clip planes.

Hence, in our basic camera class, we will need getters/setters for our four parameters and a function to calculate the view and projection matrices. Although the camera class in games would need more functions such as pitch and yaw, we will first create a base class.

Implementing the basic camera

Our base camera class has variables and basic functionality defined, which will be used in all our different camera implementations. In different types of 3D games, we generally require three types of cameras: free, target, and orbit.

- A free camera can rotate and move freely in any direction. This camera type is used in games where you want to look at the whole scene without any particular context or target. It is mostly used as a first-person camera where you want to use the camera to view the scene as if the player was viewing it.
- A target camera, though, can move freely in any direction but it will always face a target. The target is generally defined by a vector. This camera is generally used as a third-person camera. We want to see the player moving in the scene; hence, the camera follows the person.
- An orbit camera is a specific type of target camera that can rotate in any direction around the target and allows for limited rolling.

We have implemented our base class in `camera.js` present in the `primitive` directory. Open the file in your favorite text editor and let's walk through the code to understand it:

```
Camera =function ()
{
  // Raw Position Values
  this.left = vec3.fromValues(1.0, 0.0, 0.0); // Camera Left vector
  this.up = vec3.fromValues(0.0, 1.0, 0.0); // Camera Up vector
  this.dir = vec3.fromValues(0.0, 0.0, 1.0); // The direction its
looking at
  this.pos = vec3.fromValues(0.0, 0.0, 0.0); // Camera eye position
  this.projectionTransform = null;
  this.projMatrix;
  this.viewMatrix;
  this.fieldOfView = 55;
  this.nearClippingPlane =0.1;
  this.farClippingPlane = 1000.0;
};
```

In the preceding constructor, we have declared and initialized all the properties that we need to create the view and projection matrices.

The following code lists all the getters for the up, left, and position vectors. Also,

we have created getters to access the computed projection and view matrix.

```
Camera.prototype.getLeft =function ()
{
  return vec3.clone(this.left);
}
Camera.prototype.getPosition =function ()
{
  return vec3.clone(this.pos);
}
Camera.prototype.getProjectionMatrix =function ()
{
  return mat4.clone(this.projMatrix);
}
    Camera.prototype.getViewMatrix =function ()
{
  return mat4.clone(this.viewMatrix);
}
Camera.prototype.getUp =function ()
{
  return vec3.clone(this.up);
}
```

The following code lists the getters for field of view and clipping planes. The field of view getter returns the value in degrees.

```
Camera.prototype.getNearClippingPlane =function ()
{
  return this.nearClippingPlane;
}
Camera.prototype.getFieldOfView =function ()
{
  return this.fieldOfView;
}
```

The next set of functions are setters for the clipping planes and field of view. The near clip plane has to be a positive value. The field of view value has to be between 0 and 180 degrees. The far clipping plane should not be set to an extremely large value. This can create depth buffer precision problems such as z-fighting. For more information about the depth buffer, refer to http://www.opengl.org/resources/faq/technical/depthbuffer.htm.

```
Camera.prototype.setFarClippingPlane =function (fcp)
{
  if (fcp > 0)
  {
```

```
    this.farClippingPlane = fcp;
  }
}
Camera.prototype.setFieldOfView =function (fov)
{
  if (fov > 0 && fov < 180)
  {
    this.fieldOfView = fov;
  }
}

Camera.prototype.setNearClippingPlane =function (ncp)
{
  if (ncp > 0)
  {
    this.nearClippingPlane = ncp;
  }
}
```

The `apply` function takes the aspect ratio as a parameter and computes the matrices, as shown in the following code:

```
Camera.prototype.apply =function (aspectRatio)
{
  var matView=mat4.create();
  var lookAtPosition=vec3.create();
  vec3.add(lookAtPosition,this.pos, this.dir);
  mat4.lookAt(matView, this.pos, lookAtPosition, this.up);
  mat4.translate(matView,matView,vec3.fromValues(-this.pos[0], -
this.pos[1], -this.pos[2]));
  this.viewMatrix = matView;
  // Create a projection matrix and store it inside a globally
accessible place.
  this.projMatrix=mat4.create();
  mat4.perspective(this.projMatrix, degToRadian(this.fieldOfView),
aspectRatio, this.nearClippingPlane, this.farClippingPlane)));
}
```

Our basic camera does not take the `lookAtPosition` (target object's position) value but the `lookAt` function of `glMatrix` requires one. Hence, we first calculate the `lookAtPosition` value from the position and the direction vectors by adding them up. Then, we invoke the `lookAt` function to calculate our view matrix. The projection matrix is calculated using the `perspective` function of `glMatrix`.

Understanding the free camera

The `FreeCamera` class that we are going to write will be controlled by the keyboard. We would like to move forward, backward, left, right, and rotate on its axes. Let's first define the actions that we want our free camera to perform with the corresponding key combinations. The key combinations and the respective actions to be performed are shown in the following table:

Key combination	**Action to be performed**
Left arrow	Move to the left along the *x* axis
Right arrow	Move to the right along the *x* axis
S	Move to the left along the *y* axis
W	Move to the right along the *y* axis
Up arrow	Move to the left along the *z* axis
Down arrow	Move to the right along the *z* axis
Shift + *A*	Tilt to the left
Shift + left arrow	Rotate counter clockwise
Shift + *D*	Tilt to the right
Shift + right arrow	Rotate clockwise
Shift + up arrow	Look down

Shift + down arrow	Look up

We don't want the users of the `FreeCamera` class to set the up, left, and direction vectors either. It would be very confusing for the user to provide the value for these vectors. Hence, we decided that the user would provide two settings to our camera class:

- **Position**: This setting will define the position of the camera.
- **"look at" position**: This setting will define the point on the scene that the camera looks at. If we set the "look at" point, then the behavior becomes more like a target camera. We have explained how other vectors are adjusted when we set the "look at" position.

We will calculate the up, left, and direction vectors using the preceding setting values.

Also, we have added two more properties to the `FreeCamera` class: the angular and linear velocities. Although we will not use these parameters in our present implementation of the camera class, they will be very useful when we discuss animations in Chapter 6, *Applying Textures and Simple Animations to Our Scene*.

Implementing the free camera

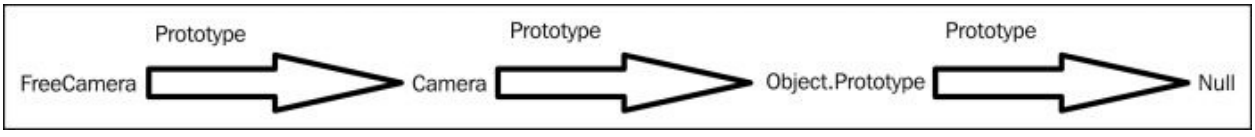
Open `freecamera.js` in your favorite text editor. The first step is to inherit the `Camera` class in `FreeCamera` so that it gets access to basic functions, such as apply, and all the properties such as up, left, direction, position, and FOV that are important for any type of camera.

```
FreeCamera = inherit(Camera, function ()
{
  superc(this);
  // Delta Values for Animations
  this.linVel = vec3.fromValues(0.0, 0.0, 0.0); // Animation of
positions
  this.angVel = vec3.fromValues(0.0, 0.0, 0.0); // Animations of
rotation around (side Vector, up Vector, dir Vector)
});
```

The preceding code uses the `inherit` function defined in the `utils.js` file present in the `primitive` directory. The `inherit` function uses JavaScript prototyping for inheritance. Let's touch upon the inheritance of the `Camera` class quickly. Open `utils.js` in your favorite text editor. In the file, you will find two functions: `inherit` and `superc`. The `inherit` function takes the name of the parent class and the constructor of the child class as parameters and uses prototyping for inheritance.

JavaScript is a prototype-based language, which means that every object can be a prototype of another object. The process forms a prototype chain, which is a linked list of objects, from which an object inherits the properties and methods of its parent object.

A nonstandard magical property, `__proto__`, is a mimic of the internal `Prototype` property. We can modify `__proto__` to dynamically change the prototype chain of an existing object. See the following figure to understand the hierarchy:



The following code shows how we break the chain by setting the `__proto__` property:

```
function inherit(parentObject, child)
  {
    child.prototype.__proto__ = parentObject.prototype;
    child.prototype.__parent = parentObject;
    return child;
  }
```

Then, the `superc` function invokes the `apply` prototype function. It is invoked by the constructor of the child class. Using the `apply` function, we can write a method once, and then inherit it in another object, without having to rewrite the method for the new object. The `superc` function is defined as follows:

```
function superc(obj)
  {
    var tmpparent = obj.__parent;
    // Temporarily change our parent to be our parent's parent to
    // avoid infinite recursion.
    if (!(obj.__parent===undefined) &&
o.__parent.prototype.__parent) {
      obj.__parent = obj.__parent.prototype.__parent;
    }
    tmpparent.prototype.constructor.apply(obj);
    delete obj.__parent;
  }
```

We won't discuss more JavaScript inheritance here, but you can read about it at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain.

We initialize two properties: `lineVel` (linear velocity) and `angVel` (angular velocity) in the constructor. Now, let's discuss the implementation of our two most important initialization functions: `setLookAtPoint` and `setPosition`. The `setPosition` function is straightforward. It takes an array of the [x, y, z] location coordinates of the camera, creates a new vector, and initializes our `pos` variable inherited from the parent `Camera` class.

By adjusting the position of the free camera, the camera's rotational angles are not affected. Hence, we can simply copy the values of the `newVec` variable to its position (`pos`) class variable, as shown in the following code snippet:

```
/**
```

```
 Set the new location of the camera.
 @param {Array} newVec An absolute value of where to
 place the camera.
 */
    FreeCamera.prototype.setPosition =function (newVec)
{
  this.pos=vec3.fromValues(newVec[0],newVec[1],newVec[2])
}
```

The `setLookAtPoint` function uses the position vector and uses point (`newVec`) values to calculate the up, left, and direction vectors. The code starts with a check to see whether the position or `lookAtPoint` is at the world origin. If it is, it should not do anything, but simply use the values for up [0, 1, 0], left [1, 0, 0], and dir [0, 0, 1] from the parent constructor. If any of the values are different, then calculate the direction vector by subtracting the value of the position from the look at point vector (`vec3.subtract(this.dir, newVec, this.pos)`) and normalizing the direction vector (`vec3.normalize(this.dir, this.dir)`) to a unit vector.

Then, we adjust our left and up vectors. We calculate the left vector assuming that the up vector is [0,1,0] and compute the cross product between the direction and up vectors (`vec3.cross(this.left, vec3.fromValues(0, 1, 0), this.dir )`). Then, we calculate the up vector by computing the cross product of the left and direction vectors (`vec3.cross(this.up, this.dir, this.left)`).

All the mathematics involved is based on the fact that all the three vectors are perpendicular to each other, that is, they are orthogonal. If we have two vectors, we can calculate the third by calculating the cross product of the two vectors. The cross product yields a vector that is perpendicular to the original two vectors. The `setLookAtPoint` function is defined as follows:

```
FreeCamera.prototype.setLookAtPoint =function (newVec)
{
    // if the position hasn't yet been changed and they want the
    // camera to look at [0,0,0], that will create a problem.
  if (isVectorEqual(this.pos, [0, 0, 0]) && isVectorEqual(newVec,
[0, 0, 0]))
  {
  }
  else
  {
    // Figure out the direction of the point we are looking at.
```

```
    vec3.subtract(this.dir, newVec, this.pos);
     vec3.normalize(this.dir, this.dir);
    // Adjust the Up and Left vectors accordingly
    vec3.cross(this.left, vec3.fromValues(0, 1, 0), this.dir );
    vec3.normalize(this.left, this.left);
    vec3.cross(this.up, this.dir, this.left);
    vec3.normalize(this.up, this.up);
  }
}
```

The next very important function is `rotateOnAxis`; this function is the base function for pitch, yaw, and roll functionalities. It takes two parameters: the axis vector (`axisVec`) and an angle in radians. The axis vector holds the axis (up, left, and direction) on which we want the rotation. In this function, we will not use the rotation matrix but use quaternions described earlier. We will use the `setAxisAngle` function of our `quat` class to compute the rotation along an axis and store it in our `quate` variable: `quat.setAxisAngle(quate, axisVec, angle)`. Then, we transform our three vectors with the rotation quaternion. Rotation using quaternions is much faster than using the rotation matrix, as the rotation matrix would use 9/16 elements but quaternions use four elements (`i`, `j`, `k`, and `w`). The `rotateOnAxis` function is defined as follows:

```
FreeCamera.prototype.rotateOnAxis =function (axisVec, angle)
{
  // Create a proper Quaternion based on location and angle
  var quate=quat.create();
  quat.setAxisAngle(quate, axisVec, angle)
  // Create a rotation Matrix out of this quaternion
  vec3.transformQuat(this.dir, this.dir, quate)
  vec3.transformQuat(this.left, this.left, quate)
  vec3.transformQuat(this.up, this.up, quate)
  vec3.normalize(this.up, this.up);
  vec3.normalize(this.left, this.left);
  vec3.normalize(this.dir, this.dir);
}
```

For functionalities such as pitch, yaw, and roll, we will invoke the `rotateOnAxis` function to calculate our new vectors. For `pitch`, we will pass the left vector as a parameter (`axisVec`) to `rotateOnAxis`, as pitch is rotation around the *x* axis. For `yaw`, we will pass the up vector, as yaw is rotation around the *y* axis, and for `roll`, we will pass the direction vector for roll. These three functions are defined in the following code snippet:

```
FreeCamera.prototype.yaw =function (angle) {
```

```
  this.rotateOnAxis(this.up, angle);
}
FreeCamera.prototype.pitch =function (angle) {
  this.rotateOnAxis(this.left, angle);
}
FreeCamera.prototype.roll =function (angle) {
  this.rotateOnAxis(this.dir, angle);
}
```

We also added a `moveForward` function. It moves the camera forward in the direction it is pointing. We pass the distance as the parameter(s) and then multiply our direction vector with this scalar and add or subtract the result from the position. The `moveForward` function is defined as follows:

```
FreeCamera.prototype.moveForward = function(s) {
  var newPosition = [this.pos[0] - s*this.dir[0], this.pos[1] -
s*this.dir[1], this.pos[2] - s*this.dir[2]];
  this.setPosition(newPosition);
}
```

The next set of functions will be used in animation in Chapter 6, *Applying Textures and Simple Animations to Our Scene*. The objective of these functions is to move our camera at a constant rate with the scene or use it as a first-person camera to move the camera along with the model. These functions either move or rotate the camera at a fixed or angular velocity, as shown in the following code:

```
FreeCamera.prototype.setAngularVel =function (newVec){
  this.angVel[0] = newVec[0];
  this.angVel[1] = newVec[1];
  this.angVel[2] = newVec[2];
}

FreeCamera.prototype.getAngularVel =function (){
  return vec3.clone(this.angVel);
}

FreeCamera.prototype.getLinearVel =function (){
  return vec3.clone(this.linVel);
}

FreeCamera.prototype.setLinearVel =function (newVec){
  this.linVel[0] = newVec[0];
  this.linVel[1] = newVec[1];
  this.linVel[2] = newVec[2];
```

```
}

FreeCamera.prototype.update =function (timeStep)
{
  if (vec3.squaredLength(this.linVel) == 0 &&
vec3.squaredLength(this.angularVel) == 0)
  return false;
  if (vec3.squaredLength(this.linVel) > 0.0)
  {
    // Add a velocity to the position
    vec3.scale(this.velVec, this.velVec, timeStep);
    vec3.add(this.pos, this.velVec, this.pos);
  }
  if (vec3.squaredLength(this.angVel) > 0.0)
  {
    // Apply some rotations to the orientation from the angular
velocity
    this.pitch(this.angVel[0]  timeStep);
    this.yaw(this.angVel[1]  timeStep);
    this.roll(this.angVel[2] * timeStep);
  }
  return true;
}
```

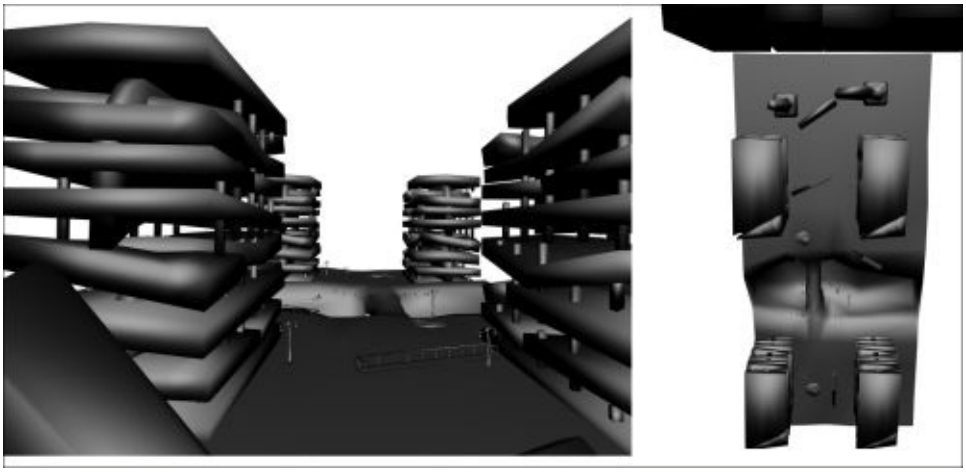
Note

The preceding free camera code does not calculate the view matrix. Each functionality just updates our vectors. We calculate our view matrix in the `apply` function defined in the parent class.

Now, let's use our free camera in the scene that we created in Chapter 3, *Loading the Game Scene*.

Using our free camera

We want to view the scene from Chapter 3, *Loading the Game Scene*, from different angles and positions as shown in the following screenshot:



Here, on the left-hand side, is the view from the city line, parallel to the road, and to the right-hand side is the top view.

Now, let's add a camera to our scene. Open `05-Loading-Scene-With-Camera.html` in your favorite text editor. There is no change in our shader code.

Implementing the control code

The first change is in our `start` function. We have initialized our camera object in the function before adding objects to the scene. We set the position and the "look at point" position of the camera. You will have to change the values in accordance to the scene you plan to develop for your game, as shown in the following code snippet:

```
cam=new FreeCamera();
cam.setPosition(new Array(0.0, 25.0, 150.0));
cam.setLookAtPoint(vec3.fromValues(0.0, 0.0, 1.0));
```

Before we understand the other changes, let's first understand the order of multiplication of the perspective, model, and view transformations. First the view matrix is multiplied with the model transformation matrix, *Mv = V*M*.

Then, the projection matrix is multiplied with the Mv matrix, *MvP = P*V*M*.

The vertices are transformed in the vertex shader:

```
vec4 vertexPos4 = mVMatrix  vec4(aVertexPosition, 1.0);
gl_Position= pMatrix vertexPos4;
```

Hence, the complete transformation of vertices happens in the order *MVP* vertex = P*V*M*Vertex*.

We initialize our `mvMatrix` as an identity matrix in the `initScene` function, as follows:

```
mat4.identity(mvMatrix);
```

In the `calculateMvMatrix` function that we added, we invoke the camera's `apply` function to calculate and set our view and projection matrices. We pass the aspect ratio of our scene (`gl.width`/`gl.height`) to the `apply` function. Then, we multiply our `cam.viewMatrix` with our `mvMatrix` (identity matrix). As multiplying by an identity matrix has no effect, our `mvMatrix` now holds the view matrix:

```
function calculateMvMatrix(){
        cam.apply(gl.viewportWidth / gl.viewportHeight);
        mat4.multiply(mvMatrix, mvMatrix, cam.viewMatrix);
    }
```

We invoke the `calculateMvMatrix` function after saving our `mvMatrix` on the stack. Although our `mvMatrix` only holds the identity matrix for now, it is always a good practice to store the `mvMatrix` on the stack before any operation to save its state.

```
function drawScene() {
...
  setLightUniform();
  pushMatrix();
  calculateMvMatrix();
  for(var i=0; i<stage.stageObjects.length; ++i){
...
    pushMatrix();
    mat4.translate(mvMatrix, mvMatrix,
stage.stageObjects[i].location);

    mat4.rotateX(mvMatrix, mvMatrix,
stage.stageObjects[i].rotationX);
    mat4.rotateY(mvMatrix, mvMatrix,
stage.stageObjects[i].rotationY);
    mat4.rotateZ(mvMatrix, mvMatrix,
stage.stageObjects[i].rotationZ);

    setMatrixUniforms();
...
```

```
    gl.drawElements(gl.TRIANGLES,
stage.stageObjects[i].geometry.indices.length, gl.UNSIGNED_SHORT,
0);
    popMatrix();
  }
  popMatrix();
}
```

In the preceding `drawScene` function, we have only shown the lines from code that are of relevance here. To start with, we push the `mvMatrix` on the stack, then compute the `mvMatrix` to hold the view matrix. Before transforming each scene object, we store the `mvMatrix` on the stack, then apply our model transformation on the view matrix. Keep in mind that we maintain the order by first multiplying the model matrix with the view matrix. Then, we invoke the `drawElements` call to transform the vertices with the new `mvMatrix` in the shader code. Then, pop the view matrix from the stack. When all objects are rendered, we pop our identity matrix from the matrixStack array to reinitialize our view matrix.

Hence, when we compute the view matrix, each time we invoke `drawScene`, the computed view matrix is stored on the stack and is used for each model transformation in the loop.

The last change is in `setMatrixUniforms`, where we use our projection matrix from the camera object.

```
function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false,
cam.projMatrix);
...
}
```

Adding keyboard and mouse interactions

Generally we move our characters using the mouse and keyboard, but in our case, we will rotate or move the camera instead. Keyboard interactions are easier to build compared to mouse interactions. In mouse interactions, we have to calculate the angle of rotation based on the mouse movement on the 2D screen, while in the case of the keyboard, we have to directly modify our camera position with each key press. When a key is pressed, we move the camera by multiplying `pi` with a constant factor (`this.cam.roll(-Math.PI * 0.025)`) for each step.

So, let's quickly look at the `KeyBoardInteractor.js` file present in the `primitive` directory:

```
function KeyBoardInteractor(camera, canvas){
  this.cam = camera;
  this.canvas = canvas;
  this.setUp();
}
KeyBoardInteractor.prototype.setUp=function(){
  var self=this;
  this.canvas.onkeydown = function(event) {
    self.handleKeys(event);
  }
}
```

The constructor of the `KeyBoardInteractor` class takes the `canvas` and `camera` objects as parameters. It invokes the `setup` function, which attaches a key handler to our canvas object, in our case `handleKeys`.

The `handleKeys` handler is pretty simple. If the keyboard keys are pressed along with the *Shift* key, then the handler invokes the `yaw`, `pitch`, or `roll` function of our camera object, rotating the corresponding vectors with a constant value (`this.cam.roll(-Math.PI * 0.025)`). If the *Shift* key is not pressed, then the handler gets the current position of the camera (`var pos = this.cam.getPosition()`) and then changes it by adding or subtracting the respective constant value of the *x*, *y*, and *z* axes of the position property of the camera object (`this.cam.setPosition([pos[0]+10, pos[1], pos[2]])`; here, `10` indicates the changed *x* value of the camera so as to move along positive *x*

axis).

```
KeyBoardInteractor.prototype.handleKeys=function (event) {
  if(event.shiftKey) {
    switch(event.keyCode) {//determine the key pressed
      case 65://a key
             this.cam.roll(-Math.PI  0.025);//tilt to the left
             break;
      case 37://left arrow
              this.cam.yaw(Math.PI  0.025);//rotate to the left
              break;
      case 68://d key
              this.cam.roll(Math.PI  0.025);//tilt to the right
              break;
      case 39://right arrow
              this.cam.yaw(-Math.PI  0.025);//rotate to the right
              break;
      case 83://s key
      case 40://down arrow
              this.cam.pitch(Math.PI  0.025);//look down
              break;
      case 87://w key
      case 38://up arrow
              this.cam.pitch(-Math.PI  0.025);//look up
              break;
    }
  }
  else {
    var pos = this.cam.getPosition();
    switch(event.keyCode) {//determine the key pressed
      case 65://a key
      case 37://left arrow
              this.cam.setPosition([pos[0]-10, pos[1],
pos[2]]);//move - along the X axis
              break;
      case 68://d key
      case 39://right arrow
              this.cam.setPosition([pos[0]+10, pos[1],
pos[2]]);//more + along the X axis
              break;
      case 83://s key
              this.cam.setPosition([pos[0],pos[1]-10,
pos[2]]);//move - along the Y axis (down)
              break;
      case 40://down arrow
              this.cam.setPosition([pos[0], pos[1],
pos[2]+10]);//move + on the Z axis
```

```
                break;
        case 87://w key
                this.cam.setPosition([pos[0], pos[1]+10,
pos[2]]);//move + on the Y axis (up)
                break;
        case 38://up arrow
                this.cam.setPosition([pos[0], pos[1],
pos[2]-10]);//move - on the Z axis
                break;
    }
  }
}
```

Handling mouse events

The mouse interaction is a little tricky as we have to derive a formula to calculate the angle rotation from the distance covered by the mouse when it is dragged. Open `primitive/MouseInteractor.js` in your favorite text editor and take a look at the following code:

```
MouseInteractor.prototype.mousePosX=function(x){
  return 2  (x / this.canvas.width) - 1;
}
MouseInteractor.prototype.mousePosY=function(y){
  return 2  (y / this.canvas.height) - 1;
}
```

Our formula is simple. Our current camera angle is a function of the mouse position and the canvas size, as shown in the following code snippet:

```
MouseInteractor.prototype.setUp=function(){
  var self=this;
  this.canvas.onmousedown = function(ev) {
    self.onMouseDown(ev);
  }
  this.canvas.onmouseup = function(ev) {
    self.onMouseUp(ev);
  }
  this.canvas.onmousemove = function(ev) {
    self.onMouseMove(ev);
  }
}
MouseInteractor.prototype.onMouseUp = function(evnt){
  this.dragging = false;
}
MouseInteractor.prototype.onMouseDown = function(evnt){
  this.dragging = true;
  this.x = evnt.clientX;
  this.y = evnt.clientY;
  this.button = evnt.button;
}
```

The `setup` function initializes event handlers for the `onMouseUp`, `onMouseDown`, and `onMouseMove` events. The `onMouseDown` event enables the `dragging` flag and `onMouseUp` disables it. The `onMouseDown` event also stores the `x` and `y` coordinates in the `x` and `y` class variables. These variables are used to calculate the delta, the difference in the current, and previous mouse positions in the

`onMouseMove` event.

```
MouseInteractor.prototype.onMouseMove = function(event){
  this.lastX = this.x;
  this.lastY = this.y;
  this.x = event.clientX;
  this.y = event.clientY;
  if (!this.dragging) return;
  this.shift = event.shiftKey;
  if (this.button == 0) {
    if(this.shift){
       var dx=this.mousePosX(this.x) -this.mousePosX(this.lastX)
       var dy=this.mousePosY(this.y) -this.mousePosY(this.lastY)
       this.rotate(dx,dy);
    }
    else{
      var dy = this.y - this.lastY;
      this.translate(dy);
    }
  }
}
```

The mouse move action is defined in the `onMouseMove` event. It stores the class variables' `this.x` and `this.y` values in `this.lastX` and `this.lastY` and then retrieves the current *x* and *y* coordinates. The code then checks whether `dragging` is enabled. If enabled, it checks to see if the *Shift* key is pressed. If *Shift* is pressed, it invokes the `rotate` function, otherwise it invokes the `translate` function.

The delta rotation angle along the *x* and *y* axes is the difference of the angle calculated at the two mouse locations (last and current).

In our code, translation is the function of difference of the mouse movement along the *y* axis of the canvas; we can always improve the function to involve movement along the *x* axis. The `translate` function is defined as follows:

```
MouseInteractor.prototype.translate = function(value) {
  var translateSensitivity=30  this.SENSITIVITY;
  var c = this.camera;
  var dv = translateSensitivity  value / this.canvas.height;
  c.moveForward(dv);
}
```

The `translate` function multiplies the delta value by a translate sensitivity

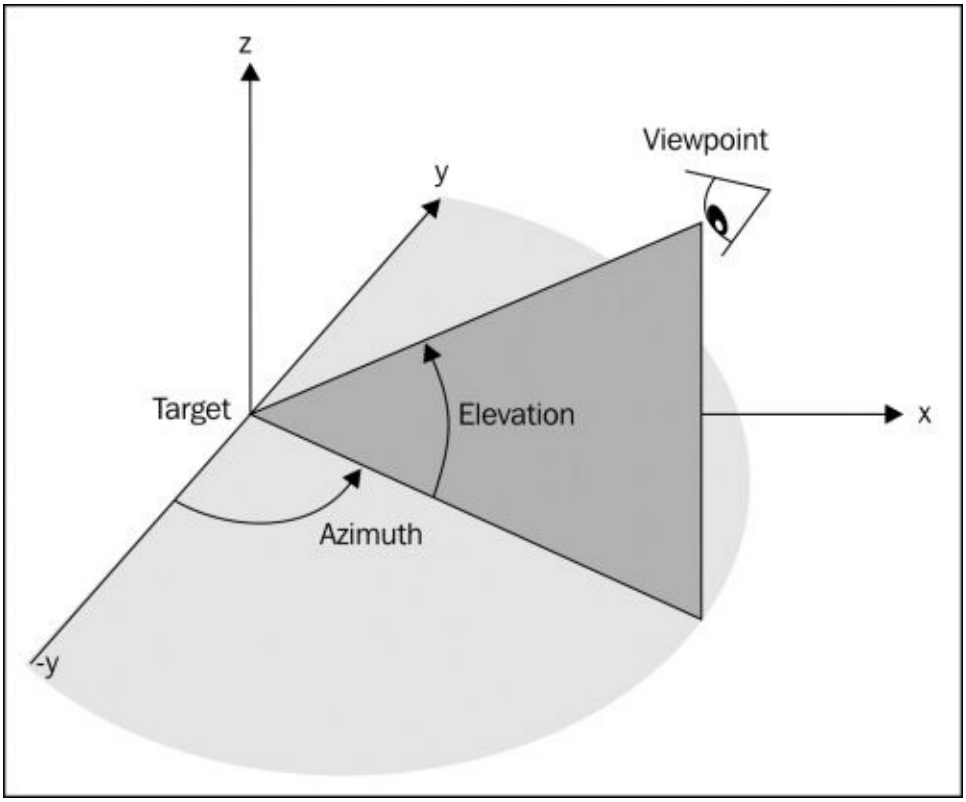
factor and then calculates the final `dv` value by dividing it with the canvas height. We used a generic function to calculate the camera translation from the mouse delta movement. You can use any sensitivity values and evolve any function that suits your game. The `rotate` function is defined as follows:

```
MouseInteractor.prototype.rotate = function(dx, dy) {
  var camera = this.camera;
  camera.yaw(this.SENSITIVITY*dx);
  camera.pitch(this.SENSITIVITY*dy);
}
```

The `rotate` function simply invokes the `Camera` class's `yaw` and `pitch` functions to rotate the camera along the *x* and *y* axes.

Comprehending the orbit camera

A camera that rotates around the target and looks at it at all times is called an orbit camera. An orbit camera can also be understood as a camera that moves along the surface of a sphere. Hence, the location of a camera is defined by three parameters: **Azimuth**, **Elevation**, and **Radius**, as shown in the following figure:



However, in our code, we will not use the terms Azimuth and Elevation, but will stick to using yaw and pitch. We do so as Azimuth and Elevation are absolute angles but yaw and pitch use incremental angles.

The third parameter is the radius. We use the term "distance" instead. In orbit cameras, we generally define boundaries by using the minimum and maximum distances from the target.

Let's understand the orbit camera better through code in the following sections.

Implementing the orbit camera

For the calculations of the orbit camera, remember the rule that the left vector of the orbit camera will always be perpendicular to the global *y* axis (that is, the global up vector).

Open `primitive/orbitcamera.js` in your favorite text editor. In the implementation of the orbit camera, we have added three new parameters:

- `closestDistance`: This parameter defines the closest the camera can go to the orbit point.
- `FarthestDistance`: This parameter defines the farthest distance the camera can go away from the orbit point.
- `orbitPoint`: If you remember, in our free camera, we only used `lookAtPoint` to calculate the initial vector (up, dir, and left). In the case of the orbit camera, we will need to store the orbit point and use it throughout to calculate all the vectors.

```
OrbitCamera = inherit(Camera, function ()
{
  superc(this);
  // this value cannot be set to less than 0.
  this.closestDistance = 0;
  // typically larger than the closest distance, but set to
  // 0 here since the user will likely overwrite it anyway.
  // this value will always be greater or equal to the closest
  // distance.
  this.farthestDistance = 0;
  // the point in space the camera will 'orbit'.
  this.orbitPoint = vec3.fromValues(0, 0, 0);
});
```

The next function is `getDistance`; it returns the distance between the orbit point and position, as shown in the following code snippet:

```
OrbitCamera.prototype.getDistance = function (){
  return vec3.distance(this.pos, this.orbitPoint);
}
```

The `goCloser` function takes the `distance` parameter and moves the camera closer to the orbit point. We move the camera towards the orbit point relative to the position of the camera. The value of `vec3.scale(shiftAmt,this.dir, distance)` is basically the value we get by multiplying the direction vector with

the distance. The camera will not move if we attempt to move it closer than the closest allowed distance. A negative value for `goCloser` could be allowed and would mean moving farther using a positive value, but this could create some confusion and is therefore not permitted. The `goCloser` function is defined as follows:

```
OrbitCamera.prototype.goCloser = function (distance)
{
  if (distance > 0)
  {
    // scale it
    var shiftAmt =vec3.create();
    vec3.scale(shiftAmt,this.dir, distance);
    var renameMe =vec3.create();
     vec3.subtract(renameMe,this.pos, this.orbitPoint);
    var maxMoveCloser = vec3.length(renameMe) -
this.getClosestDistance();

    if (vec3.length(shiftAmt) <= maxMoveCloser)
    {
       vec3.add(this.pos,this.pos, shiftAmt);
      return true;
    }
  }
  return false;
}
```

Similar to `goCloser`, the `goFarther` function sets the camera at a new radius (distance) and position. What is different from `goCloser` is that in `goFarther` we first multiply the direction vector with a negative scalar (`vec3.scale(negativeDist, this.dir, -1)`), and then we multiply the new direction vector with the distance scalar (`vec3.scale(shiftAmt, negativeDist, distance)`). Then, we calculate the new position by adding the calculated vector (`shiftAmt`) to the position, after checking whether the new position (`newpos`) exceeds the maximum distance. The `goFarther` function is defined as follows:

```
OrbitCamera.prototype.goFarther = function (distance)
{
  if (distance > 0)
  {
    //
    var shiftAmt = vec3.create();
    var negativeDist=vec3.create();
    vec3.scale(negativeDist,this.dir, -1);
```

```
    vec3.scale(shiftAmt,negativeDist, distance);
    var newpos =vec3.create();
    vec3.add(newpos,this.pos, shiftAmt);
    var distanceBetweenCamAndOP = vec3.distance(newpos,
this.orbitPoint);
    if (distanceBetweenCamAndOP <= this.getFarthestDistance())
    {
      this.pos = newpos;
      return true;
    }
  }
  return false;
}
```

The `setPosition` function is important to set the initial position of the camera. It first calculates the distance of the new position from the orbit point. The distance should be between the maximum and minimum distance. After we set the new position, we will need to update our vectors. We first calculate the direction vector by subtracting the orbit point from the position vector (`vec3.subtract(camPosToOrbitPoint, this.orbitPoint, this.pos)`).

There can be a case where the direction vector is parallel to the global *y* axis. In that case, we do not need to calculate the left vector. It will remain the same (as it will lead to a zero vector). Hence, we first find the cross product of our direction vector stored in `camPosToOrbitPoint` with the global up vector [0,1,0] (`vec3.cross(tempVec,camPosToOrbitPoint, vec3.fromValues(0, 1, 0)`). If the result is zero, we just normalize the direction vector and calculate the up vector from the left and direction vectors (`vec3.cross(this.up ,this.dir, this.left)`). However, if they are not parallel, we first calculate our left vector by multiplying the direction vector with the global up vector [0,1,0] (`vec3.cross(this.left,vec3.fromValues(0, 1, 0), this.dir)`), and then from the left and direction vector, we calculate the up vector.

```
OrbitCamera.prototype.setPosition = function (position) {
  var distFromNewPosToOP = vec3.distance(this.orbitPoint,
position);

  if (distFromNewPosToOP >= this.getClosestDistance() &&
distFromNewPosToOP <= this.getFarthestDistance()) {
    this.pos[0] = position[0];
    this.pos[1] = position[1];
    this.pos[2] = position[2];
    var camPosToOrbitPoint =vec3.create();
    vec3.subtract(camPosToOrbitPoint,this.orbitPoint, this.pos);
```

```
    vec3.subtract(camPosToOrbitPoint,this.orbitPoint, this.pos);

    var tempVec=vec3.create();
    vec3.cross(tempVec, camPosToOrbitPoint, vec3.fromValues(0, 1,
0));
    if (isVectorEqual([0, 0, 0],tempVec)) {
      vec3.normalize(this.dir ,camPosToOrbitPoint);
      vec3.cross(this.up ,this.dir, this.left);
    }
    else {
      vec3.subtract(this.dir,this.orbitPoint, this.pos);
      vec3.normalize(this.dir,this.dir);
      vec3.cross(this.left,vec3.fromValues(0, 1, 0), this.dir);
      vec3.cross(this.up,this.dir, this.left);
    }
  }
}
```

In the preceding code, our complete calculation relied on the rule that the left vector is perpendicular to the global up vector. Hence, we did not calculate the left vector when the direction vector became parallel to the global up vector [0,1,0]. However, in this case, the cross product of the global up and direction vectors would become zero. So, remember that the left vector for the orbit camera is a cross product between the global up vector and direction vector (`vec3.fromValues(0, 1, 0)`).

It is time to discuss our yaw and pitch implementations in the orbit camera. Let's talk of pitch first. The first use case is when the orbit point and camera position is the same. Then pitch will not change the left vector; it will only change the direction and up vectors. But, if the orbit points and position vectors are different, then we have to calculate all three vectors. Let's quickly look at the algorithm for the calculation of the new position and our vectors.

Understanding the pitch function for the orbit camera

We'll now see the `pitch` function for the orbit camera. If the orbit point and position are the same, perform the following steps:

1. Create a rotation quaternion around the left axis as follows:

   ```
   quat.setAxisAngle(quate, this.left, angle);
   ```

2. The position of the camera will not change because the rotation is occurring in the same place. (the orbit point and position are the same), only the

direction will change with the rotation. The transform direction with the new rotation quaternion is defined as follows:

```
vec3.transformQuat(this.dir, this.dir, quate);
```

3. Calculate the unit direction vector as follows:

```
vec3.normalize(this.dir, this.dir);
```

4. Calculate the up vector from the left and new direction vectors as follows:

```
vec3.cross(this.up, this.dir, this.left);
```

5. Normalize the up vector as follows:

```
vec3.normalize(this.up, this.up);
```

If the orbit point and camera position are different, then perform the following steps:

1. Create a rotation quaternion around the left axis as follows:

```
quat.setAxisAngle(quate, this.left, angle);
```

2. Calculate the new position after transforming the old position with the quaternion. This calculates the new position relative to the origin.

```
vec3.transformQuat(newpos,this.pos,quate);
```

3. Transform the position relative to the origin to relative to the orbit point as follows:

```
vec3.add(this.pos,newpos, this.orbitPoint);
```

4. Calculate the direction vector from the orbit point and new position and also calculate the unit direction as follows:

```
vector.(vec3.subtract(this.dir,this.orbitPoint, this.pos);
vec3.normalize(this.dir, this.dir));
```

5. Calculate the up vector using the cross product of the direction and left vectors. Compute the unit up vector and normalize it as shown in the following code:

```
vec3.cross(this.up,this.dir, this.left);
vec3.normalize(this.up,this.up));
```

6. Calculate the left vector using the cross product of the up and direction

vectors. Compute the unit left vector and normalize it as shown in the following code:

```
vec3.cross( this.left ,this.up, this.dir);
vec3.normalize(this.left ,this.left))
```

The complete pitch function is defined as follows:

```
OrbitCamera.prototype.pitch = function (angle) {
  if (isVectorEqual(this.pos, this.orbitPoint)) {
    var quate=quat.create();
    quat.setAxisAngle(quate, this.left, angle);
    vec3.transformQuat(this.dir, this.dir, quate);
    vec3.normalize(this.dir,this.dir);
    vec3.cross(this.up,this.dir, this.left);
    vec3.normalize(this.up, this.up);
  }
  else {
    vec3.subtract(this.pos,this.pos, this.orbitPoint);
    var quate =quat.create();
    quat.setAxisAngle(quate,this.left, angle);
    var newpos = vec3.create();
    vec3.transformQuat(newpos, this.pos,quate);
    vec3.add(this.pos, newpos, this.orbitPoint);
    vec3.subtract(this.dir, this.orbitPoint, this.pos);
    vec3.normalize(this.dir, this.dir);
    vec3.cross(this.up, this.dir, this.left);
    vec3.normalize(this.up, this.up);
    vec3.cross( this.left , this.up, this.dir);
    vec3.normalize(this.left , this.left);
  }
}
```

Understanding the yaw function for the orbit camera

The orbit camera's `yaw` function algorithm is as follows:

If the orbit point and position are the same, perform the following steps:

1. We calculate the rotation quaternion from the global up vector [0,1,0] and not from the camera up vector as follows:

   ```
   quat.setAxisAngle(quate,[0, 1, 0], angle);
   ```

2. The camera position does not change. We transform each vector with the

quaternion like we do in the free camera as follows:

```
vec3.transformQuat(this.dir, this.dir, quate);
vec3.transformQuat(this.left, this.left, quate);
vec3.transformQuat(this.up, this.up, quate);
```

3. Compute unit vectors for all three vectors.

If the orbit point and camera position are different, perform the following steps:

1. Create a rotation quaternion around the global up vector as follows:

```
quat.setAxisAngle(quate, [0,1,0], angle);
```

2. Calculate the new position after transforming the old position with a quaternion. This calculates a new position relative to the origin.

```
vec3.transformQuat(newpos,this.pos,quate);
```

3. Transform the position from relative to origin to relative to orbit point as follows:

```
vec3.add(this.pos,newpos, this.orbitPoint);
```

4. Calculate the direction vector from the orbit point and new position, and calculate the unit direction vector as follows:

```
vec3.subtract(this.dir,this.orbitPoint, this.pos);
vec3.normalize(this.dir,this.dir);
```

5. Calculate the up vector after transforming it with the quaternion as follows:

```
vec3.transformQuat(this.up, this.up, quate);
```

6. Calculate the left vector using the cross product of up and direction vectors. Compute the unit left vector (`vec3.cross( this.left, this.up, this.dir); vec3.normalize(this.left, this.left)`):

```
OrbitCamera.prototype.yaw = function (angle) {
  if (isVectorEqual(this.pos, this.orbitPoint)) {
    var quate=quat.create();
    quat.setAxisAngle(quate,[0, 1, 0], angle);
    vec3.transformQuat(this.dir, this.dir, quate)
    vec3.transformQuat(this.left, this.left, quate)
    vec3.transformQuat(this.up, this.up, quate)
    vec3.normalize(this.up,this.up);
    vec3.normalize(this.left,this.left);
    vec3.normalize(this.dir,this.dir);
```

```
    }
    else {
      var camPosOrbit =vec3.create();
      vec3.subtract(camPosOrbit,this.pos, this.orbitPoint);
      var quate=quat.create();
      quat.setAxisAngle(quate,[0, 1, 0], angle);
      var newpos = vec3.create();
      vec3.transformQuat(newpos, camPosOrbit, quate);
      vec3.add(this.pos,newpos, this.orbitPoint);
      vec3.subtract(this.dir,this.orbitPoint, this.pos);
      vec3.normalize(this.dir,this.dir);
      vec3.transformQuat(this.up, this.up, quate)
      vec3.normalize(this.up,this.up);
      vec3.cross(this.left,this.up, this.dir);
      vec3.normalize(this.left,this.left);
    }
  }
```

The next function is `setDistance` where we set the initial camera location. It simply copies the `orbitpoint` to `position` and then invokes `goFarther`.

```
OrbitCamera.prototype.setDistance = function (distance)
{
  if (distance >= this.getClosestDistance() && distance <=
this.getFarthestDistance())
  {
    // place the camera at the orbit point, then goFarther
    vec3.copy(this.pos,this.orbitPoint);
    this.goFarther(distance);
  }
}
```

Using an orbit camera

Open `05-Loading-Scene-OrbitCamera.html` in your favorite text editor. The only change we have in this file is to replace the free camera object with the orbit camera object. Also, set the basic properties for the orbit camera.

```
cam=new OrbitCamera();
cam.setFarthestDistance(300);
cam.setClosestDistance(60);
cam.setOrbitPoint([0.0, 20.0, 0.0]);
cam.setDistance(100);
```

We added some changes to `primitive/KeyboardInteractor.js` to handle the orbit camera.

If the camera object is of the `OrbitCamera` type, then only two keys are functional, the up and down arrow keys. We invoke the `goCloser` and `goFarther` functions of the camera object.

```
if(this.cam instanceof OrbitCamera) {
  switch(event.keyCode) {//determine the key pressed
    case 65://a key
    case 37://left arrow
            break;
    case 68://d key
    case 39://right arrow
            break;
    case 83://s key
            break;
    case 40://down arrow
            this.cam.goFarther(10);//move + on the Z axis
            break;
    case 38://up arrow
            this.cam.goCloser(10);//move - along the Y axis (down)
            break;
  }
}
```

The changes in `primitive/MouseInteractor.js` are listed as follows:

```
MouseInteractor.prototype.translate = function(value) {
...
  if(c instanceof OrbitCamera) {
    if(dv>0) {
      c.goFarther(dv);
```

```
    }
    else{
      c.goCloser(-dv);
    }
  }
  else{
    c.moveForward(dv);
  }
}
```

If the camera object is of the `OrbitCamera` type, then invoke `goCloser` or `goFarther`, otherwise invoke the `moveForward` function of `FreeCamera`.

Summary

In this chapter, we discussed the most important aspects of 3D rendering: the view and projection matrices. We created our own camera class to compute the view matrix from the up, left, direction, and position vectors. We demonstrated that in a simple concept, that for each camera operation, such as `moveForward`, yaw, pitch, and roll, the orthogonal vectors are affected and the logic to adjust these vectors depends on the type of camera we implement. We demonstrated this fact by implementing the free and orbit cameras.

We also discussed the projection matrix and implemented a perspective camera.

In the next chapter, we will discuss animations. We will apply animations on our objects and also discuss the different algorithms to generate the movement trajectory of the objects. We will use our `camera` class as a first person camera to follow moving objects in our game scene.

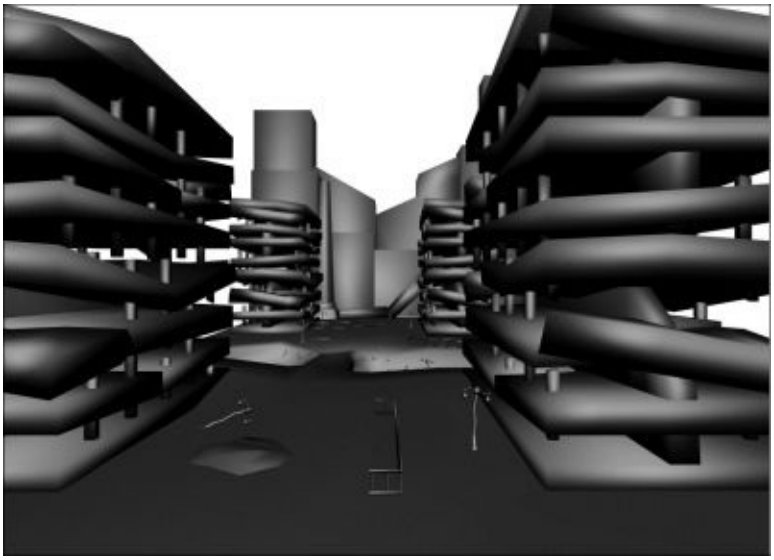
Chapter 6. Applying Textures and Simple Animations to Our Scene

Until now, we have only added static objects to our scene, but in this chapter, we will animate our objects. We need simple animations in all our games. We will start by simulating a first-person camera and take it forward by giving weapons to our character in 5000 AD. We will introduce you to different animation techniques used in game engines. Then we will apply the most common animation techniques used in HTML based games. The topics we will cover are as follows:

- Architectural update: add textures to our scene
- Animation types in 3D games
- First-person camera
- Simple bullet action: linear animation
- Multiple bullets: how to reuse objects
- Grenade action: B-spline interpolation
- Explosion effect: texture animation

Applying textures to our scene

We covered how to apply textures in Chapter 4, *Applying Textures*, but we need not apply any textures to our scene. We used diffuse colors to render objects in our scene. Our scene looked similar to the following screenshot:



We will now update our code to add textures to our scene objects so that it looks similar to the following screenshot:

Before we start making changes to the code, let's first revise a few concepts from Chapter 4, *Applying Textures*:

- To apply textures, we need another Vertex Buffer Object that holds the texture coordinates.
- Each texture coordinate for a 2D texture is represented by an (s, t) or (u, v) pair and is called a texel.
- We pass our texture to GPU memory, and then activate it using the following code:

```
gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D,
stage.textures[stage.stageObjects[i].textureIndex].texture);
  gl.uniform1i(gl.getUniformLocation(shaderProgram,
"uSampler"), 0);
```

  WebGL can handle up to 32 textures in a single `drawElements` call. In the previous code, we activate the first object and assign our texture data to that object. Then, we assign the bound texture to the `uSampler` uniform of our shader by referencing its `index` (TEXTURE0 → 0).
- We read our texture data in our fragment shader using the shader function `texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.t))`.

`vTextureCoord` is a varying vector passed from the vertex shader to the fragment shader. The following line of code is taken from the vertex shader:

```
vTextureCoord = aTextureCoord;
```

We simply assign the texture coordinate from the vertex shader to the fragment shader. Okay, why don't we pass the value directly to the fragment shader if we are not manipulating its value in the vertex shader? Well, we pass them to the vertex shader since the texture coordinates are defined per vertex. We only pass data to fragment shader that are defined per primitive.

- The number of texture coordinates must be equal to the number of vertices of an object as each texture coordinate is mapped to a vertex and vice versa.
- The converted JSON file from an OBJ file has a `uv` array and a `vertices` array. The relation between them is encoded in the `faces` array. We used an algorithm (`geometry.verticesFromFaceUvs`) to create redundant vertices and texture coordinates so that the number of vertices is equal to the number of uv coordinates.

Applying a texture to the scene

Open the `parseJSON.js` file from the `primitive` folder of the code bundle in your favorite editor.

In this file, we add the following line of code to recreate the vertices and texture coordinates:

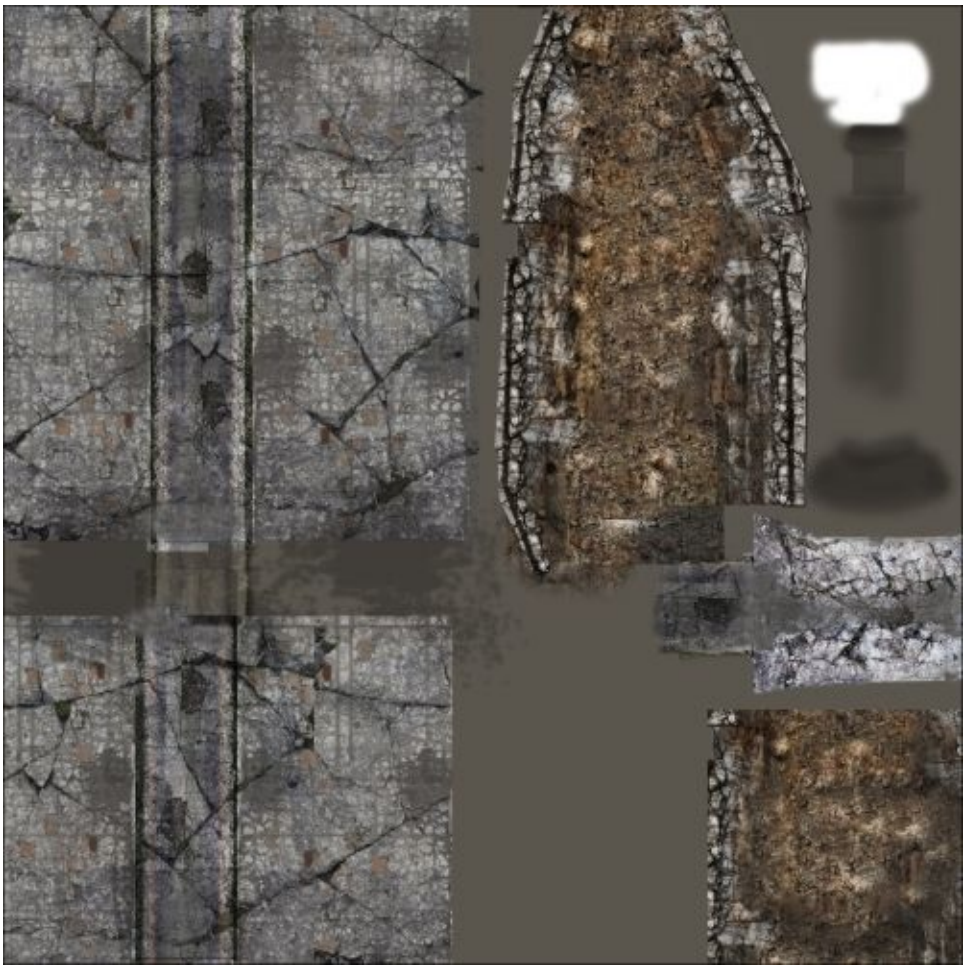
```
geometry.verticesFromFaceUvs(data.vertices,data.uvs,0);
```

We invoke this function just before creating the indices array from the `faces` array, because the previous function also reindexes the indices of vertices array.

Note

The explanation to the previous function/code is in Chapter 4, *Applying Textures*, in the *Parsing UV coordinates from the JSON file* section.

Let's first understand the basic strategy that we will follow. Most designers would prefer to create a single texture that will contain the textures of multiple objects. In our case, we have the `terrain_1024.jpg` file from the `model\obj` folder of the code bundle, as shown in the following screenshot:



The preceding screenshot has a diffuse map of four objects: terrain, railing, lamp, and dump. Why do we do this? Basically, we know that the texture file has to be in a size of power of 2, for example, 128 x 128, 128 x 64, 256 x 256, 256 x 128, and 512 x 512. If we create all files in the mentioned sizes, then we might waste precious memory space because you might have to keep empty areas in the files. Hence, in order to reduce the memory footprint of textures, we generally use this technique. Also, we can have only 32 active texture objects. For this reason as well, if we have many objects in our game, we generally create packed texture files.

We know one thing that multiple objects can have a single texture file, and we do not want to allocate space for the same texture for each object. Hence, we hold references to these textures in our `Stage` object and not the individual `StageObject`. Open the `Stage.js` file from the `primitive` folder of the code

bundle in your favorite text editor.

We have added a simple array that holds references to the texture objects using the following code:

```
this.textures=new Object();
```

We also added a new function to add the loaded texture to the stage object, as shown in the following code:

```
addTexture:function(index,name,img){
var texture=new Object();
texture.fileName=name;
texture.img=img;
texture.index=index;
texture.texture=this.gl.createTexture();
this.gl.bindTexture(this.gl.TEXTURE_2D, texture.texture);
this.gl.texImage2D(this.gl.TEXTURE_2D, 0, this.gl.RGBA,
this.gl.RGBA, this.gl.UNSIGNED_BYTE, img);
this.gl.texParameteri(this.gl.TEXTURE_2D,
this.gl.TEXTURE_MAG_FILTER, this.gl.NEAREST);
this.gl.texParameteri(this.gl.TEXTURE_2D,
this.gl.TEXTURE_MIN_FILTER, this.gl.NEAREST);
this.gl.bindTexture(this.gl.TEXTURE_2D, null);
this.textures[index]=texture;
    }
```

The preceding function takes three parameters which are as follows:

- `index`: This is a key to index the texture
- `fileName`: This is the name of the file from which the texture was loaded
- `img`: This is the pixel data of the loaded texture

In this function, we initialize our texture object and set the texture parameters. We make a new texture object as the active texture by using the `bindTexture` API call, and then load the pixel data in the `img` variable to that texture object by using the `texImage2D` function. Then, we set our filter mode to **Nearest-neighbor interpolation** using the `texParameteri` function. Refer to Chapter 4, *Applying Textures*, for an in-depth understanding of the functions that have just been mentioned. Lastly, we add our new `texture` object at the provided key (`index`) to the `textures` array.

- Open the `StageObject.js` file from the `primitive` folder of the code bundle. The individual stage object will hold the index (key) to the textures

array defined in the `StageObject` class, as shown in the following code:

```
this.textureIndex=0;
```

We also added two new class variables to the `StageObject` class. The `materialFile` class variable holds the name of the texture file and `verticesTextureBuffer` holds the reference to the vertex buffer which holds the texture coordinates. The following code defines the variables explained previously:

```
StageObject=function(){
...
this.verticesTextureBuffer=null;
this.materialFile=null;
...
};
```

After we parse our JSON data, we check whether the texture name is present in our JSON data. If present, we assign the filename to the `materialFile` property as shown in the following code:

```
StageObject.prototype.loadObject= function (data){
this.geometry=parseJSON(data);
        this.name=data.metadata.sourceFile.split(".")[0];
if(this.geometry.materials.length>0){
...
if(!(this.geometry.materials[0].mapDiffuse===undefined)){
this.materialFile=this.geometry.materials[0].mapDiffuse;
                }
...
            }
 }
```

There may be cases when no texture file is present for an object. In such cases, we use a diffuse color to render objects.

In the `StageObject` class, earlier we created two Vertex Buffer Objects (`vertices` and `normals`) and one Index Buffer Object (indices). Now, we will create another Vertex Buffer Object for UV coordinates (`this.verticesTextureBuffer`) as shown in the following code:

```
StageObject.prototype.createBuffers=function(gl){
...
if(this.materialFile!=null){
this.verticesTextureBuffer = gl.createBuffer();
```

```
this.verticesTextureBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, this.verticesTextureBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(this.geometry.uvs[0]),gl.STATIC_DRAW);
        }
...
    }
```

Open `06-Loading-Scene-With-Textures.html` in your favorite text editor. First, we will understand the changes that we need to perform in shaders in order to support both diffuse color and textures for different types of objects.

Implementing the vertex shader code

We have added three variables to our vertex shader: `attribute` (`aTextureCoord`) that references the texture coordinate array, `varying` (`vTextureCoord`) that passes these coordinates to fragment shader, and `uniform` (`hasTexture`) that sets the `varying` only if its value is true. Basically, the same vertex shader will be used for objects that use a texture as well as for objects that use only a diffuse color. We set the value of this uniform based on the value of the `materialFile` property of each `StageObject` as shown in the following code:

```
<script id="shader-vs" type="x-shader/x-vertex">
...
attribute vec2 aTextureCoord;
varying highp vec2 vTextureCoord;
uniform bool hasTexture;
void main(void) {
...
if(hasTexture){
vTextureCoord = aTextureCoord;
     }
    }
</script>
```

Implementing the fragment shader code

In the fragment shader, we add three variables: varying (`vTextureCoord`) to receive the texture coordinates from the vertex shader, uniform (`hasTexture`) to see whether the texture was associated with a primitive, and sampler (`uSampler`) which holds the reference to the then active texture. The important thing to note is that we perform our complete computation of ambient, diffuse, and specular color, but when we decide to have a texture, we do not use the material diffuse color in our computation of the `iColor` object (`+uDirectionalColor*`

directionalLightWeighting+). If we do not have a texture, we use the material diffuse color (+uDirectionalColor *materialDiffuseColor* directionalLightWeighting+). We do not use the diffuse color of the object if the object has a texture associated with it.

Lastly, we calculate the color of the fragment (gl_FragColor) by multiplying iColor with the texture color obtained from the sampler object at the vTextureCoord coordinate using the texture2DESSL function, as shown in the following code:

```
<script id="shader-fs" type="x-shader/x-fragment">
...
uniform sampler2D uSampler;
varying highp vec2 vTextureCoord;
uniform bool hasTexture;
void main(void)  {
...
if(hasTexture){
vec3iColor = (uAmbientColor*materialAmbientColor)+
(uDirectionalColor* directionalLightWeighting)+
(uSpecularColor*materialSpecularColor*specular);
gl_FragColor = vec4(iColor, 1.0)*texture2D(uSampler,
vec2(vTextureCoord.s, vTextureCoord.t));
    }
else{
vec3iColor = (uAmbientColor*materialAmbientColor)+
(uDirectionalColor materialDiffuseColor
directionalLightWeighting)+
(uSpecularColor*materialSpecularColor*specular);
gl_FragColor = vec4(iColor, 1.0);
    }
    }
</script>
```

Working with the control code

In our main control code, the first change we did was to rename addStageObject to loadStageObject and add a new addStageObject function. This was done to improve the readability of the code, and we also moved the adding of StageObject to the addStageObject function.

Let's refer to our basic strategy once again. When we load our JSON model and if the model has an associated texture file, then we first check to see whether that file has already been loaded. If it has, then we simply get its key and assign it to

the `textureIndex` property of `StageObject` and if not, then we load it, assign a unique key to it, and then associate it with the `StageObject`.

To implement this strategy, we have created a `vartextureList=[];` array. The key of the element in the array will be the name of the texture file, and the value would be the unique key to the texture object in the array (`textures`) of the `Stage` class. The following code loads the model and then its corresponding texture and assigns a texture index for the texture:

```
function loadStageObject
(url,location,rotationX,rotationY,rotationZ){
        $.getJSON(url,function(data){
Var stageObject=new StageObject();
stageObject.loadObject(data);
if(stageObject.materialFile!=null){
if((textureList[stageObject.materialFile.trim()]===undefined)){
var currentDate=new Date();
var textureIndex=currentDate.getMilliseconds();
stageObject.textureIndex=textureIndex;
textureList[stageObject.materialFile.trim()]=textureIndex;
initTextures(stageObject.materialFile.trim(),textureIndex);
          }
else{

stageObject.textureIndex=textureList[stageObject.materialFile.trim(
)];
                }
             }
addStageObject(stageObject,location,rotationX,rotationY,rotationZ);
        });
    }
```

The preceding function, `loadObject`, loads the JSON object, and then parses it in `loadObject` of the `StageObject` class (`stageObject.loadObject(data);`). If the `materialFile` variable is not null (`stageObject.materialFile!=null`), denoting that the object has an associated texture, we generate a simple, unique index using the `currentDate.getMilliseconds();` function of the JavaScript `Date` object, store it in the `textureList` object (`textureList [stageObject.materialFile.trim()] =textureIndex;`), and invoke `initTextures`. The key in the `textureList` object that references the texture object is the name of the file loaded. However, before doing this, we make sure that the key is not already defined (`textureList [stageObject.materialFile.trim()]===undefined`), denoting that the file is

not already loaded. If it is, then we simply assign the key (index) to the `StageObject(stageObject.textureIndex=textureList[stageObject.materia` object.

```
function initTextures(fileName,textureCount){
var image = new Image();
image.onload = function() {
stage.addTexture(textureCount,fileName,image); }
image.src = "model/obj/"+fileName;
    }
```

The `initTextures` function in the preceding code loads the image and passes the data to the `stage.addTexture` function that initializes the texture data and sets the filter mode.

```
function addStageObject
(stageObject,location,rotationX,rotationY,rotationZ){
cloneObjects(stageObject);
stageObject.location=location;
stageObject.rotationX=rotationX;
stageObject.rotationY=rotationY;
stageObject.rotationZ=rotationZ;
stage.addModel(stageObject);
    }
```

The `addStageObject` function in the preceding code simply sets the transformation information of the model and initializes the buffers for the object in its `addModel` function.

Next, we obtain a reference of our variables (`aTextureCoord` and `hasTexture`) defined in the shaders in the `initShaders()` function as shown in the following code:

```
function initShaders() {
...
shaderProgram.textureCoordAttribute =
gl.getAttribLocation(shaderProgram, "aTextureCoord");
gl.enableVertexAttribArray(shaderProgram.textureCoordAttribute);
shaderProgram.hasTexture = gl.getUniformLocation(shaderProgram,
"hasTexture");
...
    }
```

Finally, we modified our `drawScene` function to initialize the variables

(`hasTexture`, `uSampler` and `textureCoordAttribute`) to be passed to the shaders, as shown in the following code:

```
function drawScene() {
...
for(var i=0;i<stage.stageObjects.length;++i){
...
if(stage.stageObjects[i].materialFile!=null){
gl.uniform1i(shaderProgram.hasTexture,1);
try{
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D,
stage.textures[stage.stageObjects[i].textureIndex].texture);
gl.uniform1i(gl.getUniformLocation(shaderProgram, "uSampler"), 0);
gl.bindBuffer(gl.ARRAY_BUFFER,
stage.stageObjects[i].verticesTextureBuffer);
gl.vertexAttribPointer(shaderProgram.textureCoordAttribute, 2,
gl.FLOAT, false, 0, 0);
            }
catch(e){
console.log("Could not initialize texture buffer.");
            }
        }
else{
gl.uniform1i(shaderProgram.hasTexture,0);

        }
...
     }
...
    }
```

The preceding code iterates over all the objects, and if the `materialFile` class variable is defined from an object, then it sets the value of `hasTexture` to `1` (`true`). For active textures, we use a very simple strategy. As an object uses a single texture, we activate only one texture in the GPU (`gl.activeTexture(gl.TEXTURE0);`). Then we assign the texture from the `textures` array of the `Stage` class to the active texture buffer. The texture object is retrieved from the stage textures array using the `textureIndex` variable . The class variable `textureIndex` is initialized for each `StageObject` that is to be drawn (`stage.textures [stage.stageObjects[i].textureIndex].texture);`). This texture object is bound to the active texture `0` using the `gl.bindTexture` API call. Then, we finally assign the active texture indexed at 0 to the uniform (`uSampler`).

Well, this is all the update we need to add textures to our scene. Now let's move on to animations.

Understanding the animation types in 3D games

Animation is about adding something dynamic to our game scene, like moving objects. Until now, in all our examples, the objects were static. We simply moved our camera to have a different perspective of the scene, but the objects' location did not change over time with respect to each other. In this chapter, we will now move objects at different timings or user-generated events. We will also learn about chained animations. But before we go ahead, let's learn some basics.

In game development, the most common animation techniques are time-based animation, tweens (interpolation), and skinned animation using rigged models.

Understanding time-based animation

We will demonstrate a way of running animation and game logic at a constant speed independent of the effective frame rate. But first, let's understand what frame rate or frame-based animation is.

Understanding frame-based animation

Frame-based animation is a system in which the game world is updated, one iteration each frame. This is the easiest system to implement and we used it in our previous chapters, but it has several drawbacks. The speed of your game is effectively tied to the frame rate, which means it will run more slowly (chronologically) on older computers and faster on newer ones. This is not what is expected. So, let's go over what we did earlier.

Open the `webgl-utils.js` file from the `js` folder in your editor. The function that we have used the most in our code is listed in the following code:

```
window.requestAnimFrame = (function() {
returnwindow.requestAnimationFrame ||
window.webkitRequestAnimationFrame ||
window.mozRequestAnimationFrame ||
window.oRequestAnimationFrame ||
window.msRequestAnimationFrame ||
function(/* function FrameRequestCallback  callback,  DOMElement
Element */ element) {
window.setTimeout(callback, 1000/300);
         };
})();
```

The preceding code heavily relies on the `requestAnimationFrame` function. Let's read the definition of the function from the following link: http://msdn.microsoft.com/en-us/library/windows/apps/hh453391.aspx. It states the following:

> *The `RequestAnimationFrame` method registers a function to call when the system is ready to update (repaint) the display. This provides smoother animations and optimal power efficiency by allowing the system to determine when the animation occurs. The visibility of the web application and the display refresh rate are used to determine the appropriate speed of the animations for example, the number of frames per second supported by*

the system.

In a nutshell, the speed at which it will fire will vary from system to system. Now, let's look at how we implemented it:

```
function initScene(){
...
tick();
}
function tick(){
requestAnimFrame(tick);
drawScene();
    }
```

We invoke `tick` when the page loads, and `tick` invokes `drawScene` and invokes itself when the system is ready to update. Now, let's understand from the viewpoint of a multiplayer game where two users are in the same scene. You fire a bullet and you expect to hit because your system is fast but the other guy has not even reached the hit point (visually) as his system is slow; the scene is being drawn at a constant frame rate which varies from system to system. Even in a single user game, the timing of two events is very important. We want more of a time-based animation not frame-based, which means that if your system is slow, you should reach the hit point by dropping frames.

To put it another way, in our code, if we are updating/moving an object along the perimeter of a circle, then on all systems, the time it takes to complete the defined path should be the same. This is what we mean when we say time-based animations.

Implementing time-based animation

Open `06-Bullet-Action.html` in your text editor. In the following code, we first invoke `tick`, which in turn invokes `animate`, and `animate` invokes `drawScene`. Basically, `animate` covers for the lost time in slow systems (dropping frames/rendering them quickly). A faster frame rate does not invoke `drawScene`.

```
var rate=300;
var MINIMUM_FRAME_RATE=30;
var lastFrameTime = (new Date).getTime();
var cyclesLeftOver;
var currentTime;
var elapsedTime;
```

```
    function tick(){
requestAnimFrame(tick);
animate();
    }
function animate(){
currentTime = (new Date).getTime();
elapsedTime = currentTime - lastFrameTime;
if (elapsedTime< rate) return;
var updateIterations = Math.floor(elapsedTime /
MINIMUM_FRAME_RATE);
while(updateIterations> 0){
drawScene();
updateIterations -= 1;
        }
lastFrameTime = (new Date).getTime();
    }
```

The `animate` function is fairly straightforward. We first define `rate` and `MINIMUM_FRAME_RATE` and capture the time the page was loaded in `lastFrameTime`. We then calculate the difference since the last time `animate` was invoked and capture the result in `elapsedTime`. If `elapsedTime` is less than `rate`, we do not render the scene to accommodate for very fast systems. If the time is more, we calculate how many times the `drawScene` function was expected to be invoked during that time period by calculating `updateIterations` (`elapsedTime / MINIMUM_FRAME_RATE`). Now, we invoke `drawScene` as many times as it was supposed to be invoked during that time period.

Note

The preceding approach is not perfect as it invokes `drawScene` multiple times in a single frame, which is not ideal. We should just update our position (state) variables and invoke `drawScene` once. We will follow the updated approach in the subsequent chapters.

Comprehending interpolation

We have discussed when to update our scene, but not what to update it to, or how to update it. In most basic animations, we generally move a complete object and in this chapter, we will stick to that. So, our objective would be to move objects along a defined path, on user-generated events.

The path is defined by an equation such as y=f(x) and some discrete points, for example, if the path is along a line, then we have the equation of the line and two points: start(x,y) and end(x1,y1). Now, we need to calculate all the points along the path, which is called interpolation. So, interpolation is a method of constructing new data points within the range of a discrete set of known data points. Hence, if we have three data points such as (0,1),(1,2),(2,3), calculating the value of x which is equal to 2.5 from the mentioned set of values is called interpolation.

Linear interpolation

Linear interpolation (sometimes known as lerp) is a method of curve fitting using linear polynomials. If the two known points are given by the coordinates, then the linear interpolant is the straight line between these points. For a value *x* in the interval, the value *y* along the straight line is given from the equation *(y-y1)/(x-x1)=(y-y2)/(x-x2)*. However, this is not precise.

We will use linear interpolation in our code to define the path of a fired bullet in our example.

Polynomial interpolation

Polynomial interpolation is a generalization of linear interpolation. Note that the linear interpolant is a linear function. We now replace this interpolant by a polynomial of a higher degree. The following equation is an example of a polynomial of degree 5:

$f(x)=ax^5+bx^4+cx^3+dx^2+ex+f$

Spline interpolation

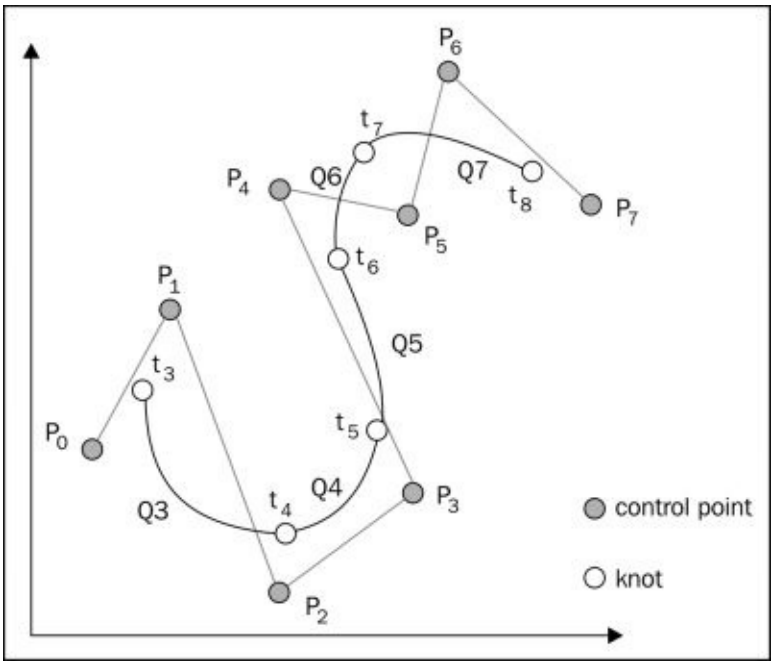
Spline interpolation is a form of interpolation where the interpolant is a special type of piecewise polynomial called a spline. Spline interpolation is preferred

over polynomial interpolation because the interpolation error can be minimized. Also, spline interpolation is better for a higher degree of polynomials because it avoids **Runge's phenomenon** (http://en.wikipedia.org/wiki/Runge's_phenomenon) found in polynomial interpolation. It is used when the function changes over the value of *x*. For example, if $x>0$, then $f(x)=x^2$, and if $x<0$, then $f(x)=1/x^2$. Piecewise polynomial refers to a different function for ranges of *x*.

Spline interpolation and linear algebra are complex topics. Refer to the link http://www.vis.uni-stuttgart.de/~kraus/LiveGraphics3D/cagd/. This web page contains good Java applets (Chapter 8, *B-spline Curves* on the web page) to understand spline interpolation.

We are not very interested in spline interpolation but very interested in its specific type called B-spline interpolation. This is the most used form of interpolation in tweening. Basically, when we want to generate a curved path, we can use this kind of interpolation.

B-spline or Basis spline is defined by its order, a set of weighted control points and a knot vector. The control points determine the shape of the curve. Typically, each point of the curve is computed by taking a weighted sum of a number of control points. However, the curve usually does not go through the control point.



In our example, we will use B-spline to generate the path of a grenade (`06-Grenade-Action.html`). We have added an implementation of B-spline interpolation in the `BSplineIterpolation.js` file from the `primitive` folder of the code bundle. We will not cover the code of its implementation, but we will discuss how to use the class.

```
var grenadeControlPoints=[[-1.5,-2,-10],[-1.5,10,-30],
[-1.5,15,-40],[-1.5,10,-60],[-1.5,0,-80]];
var splineInterpolation=new
BSplineInterpolation(grenadeControlPoints);
var grenadePositions=splineInterpolation.position;
```

In the first line of the preceding code, we defined a set of control points (`grenadeControlPoints`). Notice that for the control points, *x* (`-1.5`) remains constant and *z* (`-10`, `-30`, `-40`, `-60`, `-80`) decreases, while *y* first increases (`-2`, `10`, `15`) and then decreases (`10`, `0`). So, we are simulating a throwing action. With the grenade being thrown in the z direction, it first goes up and then comes down. The constructor of the class takes control points, interpolates all the positions between the control points, and stores the interpolated path in the position array of the class. We take all those positions and store them in our variable `grenadePositions` for future use.

Note

In a nutshell, we interpolate positions between the given points, and then change our object's location as per these points over a period of time. This creates an effect of animation. To calculate these points, we can use any interpolation technique (linear, polynomial, and B-spline).

A briefing on skinned animation

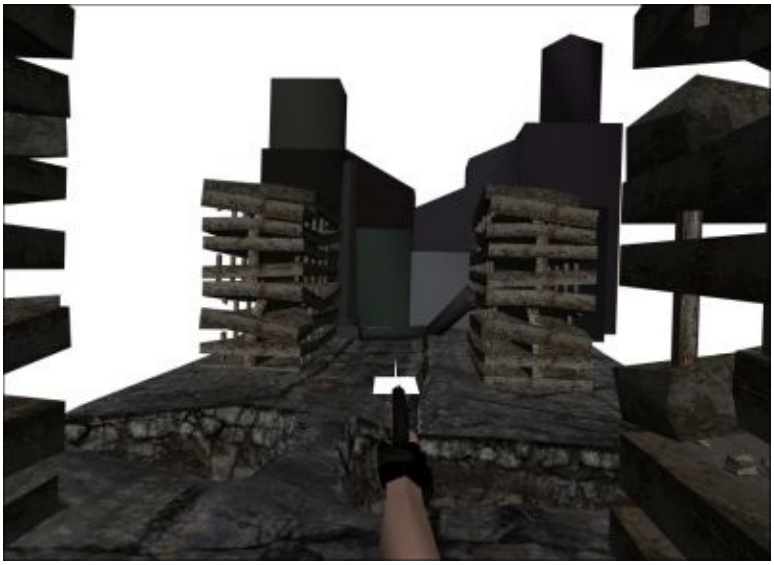
Skeletal animation is a technique in games in which a model is represented in two parts: mesh (vertices) and a hierarchical set of interconnected bones (called the skeleton or rig) that is used to animate (pose and keyframe) the mesh. Although this technique is often used for organic modeling, it only serves to make the animation process more intuitive, and the same technique can be used to control the deformation of any object—a door, a spoon, a building, or a galaxy. When we say hierarchical set, we mean that we move one parent bone, and then the subsequent child bones will also move.

We will also discuss inverse kinematics where we use the kinematics equations of a rigged model to determine the joint parameters that provide a desired position of the child bone. For example, if we have a rigged model of a human being and we move the backbone, kinematics will help us calculate the effect on each bone up to the fingers. We will discuss these animations in depth in Chapter 8, *Skinning and Animations*.

Using first-person camera

Now is the time for action! Our core objective is to create two weapons in our game: bullets and a grenade. Although we are in 5000 AD, we are still medieval and simulate the same weapons we use today. However, first we need to move around in the scene, so we need to simulate the first-person camera. This is a simple simulation of how we see a scene if we were physically present there. For example, you are sitting in a room and have a camera on a remote-controlled vehicle on the moon, and you see the surface of the moon from that camera.

We start our animation implementation with the explanation of the first-person camera as we want our bullet and grenade animations to start from the player's position. Now, to simulate this in our game, we move the camera with us in the game. The camera would simulate your eyes, it should only see what your eyes see. So, if you look straight ahead, you would not see yourself. But we needed something to show that you are moving; so, we decided to keep our model in a weaver position with a hand gun. Hence, your right arm is at your shoulder height at all times (as in the following screenshot) and our eyes can only see our own hand (a very tiring posture but we think we can live with that in our game).



Now, we need to move the camera with our arm or move the arm with our camera. We will do the latter, which means that our arm should move as the camera moves. We took this approach as we have already implemented a free

camera class in Chapter 5, *Camera and User Interaction*, and we already control it with our keys.

As discussed in Chapter 5, *Camera and User Interaction*, the camera transformation matrix is the same as any model matrix; so, if the camera was a model, then it would have the same transformation as the camera. We compute the view matrix using the `mat4.lookat` function. We also discussed in Chapter 5, *Camera and User Interaction*, in the *Converting between the camera matrix and view matrix* section, that the model matrix (camera matrix) is the inverse of the view matrix, $M=V^{-1}$.

Hence, to get the position of our arm, all we need to do is compute the camera matrix and apply the transformation to the right-hand arm object, and that is it.

```
var mMatrix=mat4.clone(cam.viewMatrix);
mat4.invert(mMatrix,mMatrix);
```

Adding the first-person camera

Open `06-Bullet-Action.html` in your text editor.

The first change we do is load our right hand model as shown in the following code:

```
function start() {
...
loadStageObject("model/weapons/rightHand.json",
[0.0,0.0,0.0],0.0,degToRadian(2),0.0);
...
}
```

We changed our `drawScene` function to apply different transformations for this model. For all other models, we will apply simple translation and rotation transformations to our ModelView matrix. But in this case, we will simply multiply the camera matrix to the ModelView matrix to position the arm exactly at the camera's location and orientation.

```
function drawScene() {
...
for(var i=0;i<stage.stageObjects.length;++i){
...
if(stage.stageObjects[i].name=="rightHand"){
var mMatrix=mat4.clone(cam.viewMatrix);
mat4.invert(mMatrix,mMatrix);
mat4.multiply(mvMatrix,mvMatrix,mMatrix);
        }
else{
mat4.translate(mvMatrix,mvMatrix, stage.stageObjects[i].location);
mat4.rotateX(mvMatrix,mvMatrix,stage.stageObjects[i].rotationX);
mat4.rotateY(mvMatrix,mvMatrix,stage.stageObjects[i].rotationY);
mat4.rotateZ(mvMatrix,mvMatrix,stage.stageObjects[i].rotationZ);
        }
...
        }
...
    }
```

We first compare if the object is `rightHand`, and then we clone the camera view matrix (`var mMatrix=mat4.clone(cam.viewMatrix)`). Then, we calculate the model matrix by calculating the inverse of the camera matrix (`mat4.invert(mMatrix,mMatrix);`). Then, we calculate our final ModelView

matrix by multiplying it with the current `mvMatrix`. (`mat4.multiply(mvMatrix,mvMatrix,mMatrix);`). Now our arm is exactly placed at our camera's position and it should look similar to the following screenshot:

However, this is not what we want. We placed our right hand right in front of our eyes/camera. Actually, our right hand is placed to the right (x=1), below (y=-3), and behind(z=-8) the camera/screen. Therefore, we need to apply a translation as shown in the following code:

```
var mMatrix=mat4.clone(cam.viewMatrix);
mat4.invert(mMatrix,mMatrix);
mat4.translate(mMatrix,mMatrix,vec3.fromValues(1,-3,-8));
mat4.multiply(mvMatrix,mvMatrix,mMatrix);
```

Hence, we moved our hand to the right (x = 1 unit), below (y = -3 units), and behind (z = -8 units) with respect to our camera.

Let's reiterate the steps that we need to perform to position any object with respect to our camera:

1. Get the camera view matrix.
2. Calculate the camera matrix by getting the inverse of the view matrix.
3. Apply transformations to the camera matrix to place the object with respect to the camera.

4. Multiply the transformed camera matrix to the ModelView matrix.

Improving the first-person camera code

The preceding code is in the `drawScene` function. If we have to apply different types of transformations to every object in our scene, then our `drawScene` function would become very big, and this is not what we want. Hence, we decided to create another code update. We added two new functions to our `StageObject` class.

Open the `StageObject.js` file from the `primitive` folder of the code bundle in your text editor.

We defined a new variable `modelMatrix` to hold the object's model matrix, as shown in the following code:

```
this.modelMatrix=mat4.create();
```

Then, we added a new function `StageObject.prototype.update`. This function is invoked by `drawScene` to calculate the model matrix, as shown in the following code:

```
StageObject.prototype.update=function(steps){
mat4.identity(this.modelMatrix);
mat4.translate(this.modelMatrix,this.modelMatrix, this.location);
mat4.rotateX(this.modelMatrix,this.modelMatrix,this.rotationX);
mat4.rotateY(this.modelMatrix,this.modelMatrix,this.rotationY);
mat4.rotateZ(this.modelMatrix,this.modelMatrix,this.rotationZ);
    }
```

We also added a new function `initialize` to the `StageObject` class. In this particular class, it does nothing but will be overridden in the child classes.

```
StageObject.prototype.initialize=function(){
         //For implementation in child class
    }
```

We also added two new variables, `this.visible` and `this.camera`, to the class.

Now let's look at our new `drawScene` functions. Open `06-Bullet-Action-Complete.html` in your editor.

```
function drawScene() {
```

```
...
for(var i=0;i<stage.stageObjects.length;++i){
if(stage.stageObjects[i].visible==true){
...
stage.stageObjects[i].update();
mat4.multiply(mvMatrix,mvMatrix,stage.stageObjects[i].modelMatrix);

...
         }
        }
...
    }
```

We iterate over our list of `stageObjects` and render only those whose `visible` property is set to `true`. We invoke the `update` function of each object to calculate the model matrix, and then multiply the model matrix with the view matrix.

We will now inherit our new `stageObject` class for every new dynamic object that we want to add to our scene. Each type of object can have different animations, and we will simply write different update functions for them.

Open the `RightHand.js` file from the `primitive/game` folder of the code bundle, in your editor. Our `RightHand` class inherits our `StageObject` class. We have overridden the `update` function of the `StageObject` class. In this function, we use the `viewmatrix` camera to calculate the model matrix. The camera object is inherited from the parent class. The following code shows our `RightHand` class derived from the `StageObject` class:

```
RightHand= inherit(StageObject, function (){
superc(this);
});
RightHand.prototype.update=function() {
var mMatrix=mat4.clone(this.camera.viewMatrix);
mat4.invert(mMatrix,mMatrix);
mat4.translate(this.modelMatrix,mMatrix,vec3.fromValues(1,-3,-8));
}
```

Open `06-Bullet-Action-Complete.html` in your editor again to accommodate our new `RightHand` class. We updated our `loadStageObject` function. Now in the function, we check the object name and create the corresponding object, like in our case, we wanted a child object for `RightHand`. We then assign a camera object to each stage object and then add an object to the `Stage` class.

```
function
```

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
        $.getJSON(url,function(data){
Var nameOfObject=data.metadata.sourceFile.split(".")[0];
var stageObject=null;
if(nameOfObject=="rightHand"){
stageObject=new RightHand();
            }
else{
stageObject=new StageObject();
            }
stageObject.camera=cam;

stageObject.loadObject(data);
...
}
}
```

Simple bullet action – linear animation

In this section, we will add another object to our scene, the bullet (`Bullet.json`), and we will move our bullet in a straight line, unlike Angelina Jolie's movie *Wanted*, where the guys curve bullets. We can do that by applying B-spline interpolation but we are still medieval and we will save it for our grenade.

The equation to move the bullet is simple: f(x)=z. We will simply change the *z* value to move the bullet. However, this is awful because if we rotate ourselves (the camera and subsequently, our arm), the bullet should go in the arm's direction. To achieve this, we need to apply the camera transformation to the bullet. Each bullet will have the same defined set of points, let's say (0,0,1), (0,0,2)...(0,0,200), but when we apply these translations to the clone of the camera matrix, then each time we fire the bullet from a different angle or position, we will get different trajectories. Actually, these defined positions are with respect to the camera and not the scene. If we apply camera transformations to the bullet, then we will get different trajectories with respect to the scene.

```
Scene Co-ordinates=Camera Matrix*Translation(Position)
```

We define positions (interpolated points) with respect to the camera, not the scene.

However, if we move our arm after firing the bullet, then the bullet should not change its initial trajectory, which means the bullet should use the initial camera matrix to calculate its transformation.

Open the `Bullet.js` file from the `primitive/game` folder in your editor.

We first inherit our `StageObject` to create the `Bullet` class. We have added three new variables in the class: `positions` to hold the calculated set of interpolated points for the animation trajectory, `initialModelMatrix` to hold the initial camera matrix so that the bullet does not change the trajectory after firing, even if the camera changes its location/orientation, and lastly, `counter` to maintain the index of the current position once the bullet is fired. Also note that we have set the `visible` property of the bullet to `false` so that the bullet is initially not visible, as shown in the following code:

```
Bullet= inherit(StageObject, function (){
superc(this);
```

```
this.visible=false;
this.positions=[];
this.initialModelMatrix=mat4.create();
this.counter=0;
});
```

We have implemented the `initialize` function of the `StageObject` class in the `Bullet` class. This function first clones the camera view matrix, and then stores the camera matrix in `initialModelMatrix`. It initializes the `position` array by invoking `calculatePositions` for a set of 200 points, and sets `counter` to `0` and `visible` to `true`, as shown in the following code:

```
Bullet.prototype.initialize=function() {
var mMatrix=mat4.clone(this.camera.viewMatrix);

mat4.invert(this.initialModelMatrix,mMatrix);
var count=200;
this.positions=[];
for(var i=0;i<count;++i){
this.positions.push(this.calculatePosition(i*this.steps));
    }
this.counter=0;
this.visible=true;
}
```

The `calculatePosition` function calculates the position by varying the z component. In the following code, our f(z) is equal to `-16-z`. We added the constant to accommodate the length of the hand with respect to the camera. Also notice the negative sign. We want the bullets to go in the negative z direction of the hand. If we rotate our hand in the world space by 180 degrees, then this value will automatically be positive with respect to the world space. The *x* and *y* remain constant with respect to the object space (in our case, the right hand).

```
Bullet.prototype.calculatePosition=function(z){
Var newPosition = [ 0,-2,-16-z];
return
vec3.fromValues(newPosition[0],newPosition[1],newPosition[2]);
}
```

Now, time for our `update` function of our bullet class. The `update` function is invoked from the `drawScene` function at a fixed time rate. Each time it is invoked, the bullet is rendered at a new position. We initialized our counter earlier. Hence, we run our bullet animation 200 for frames. The `counter` is checked against the length of the `positions` array. If the `counter` has exceeded

the length, then we alter the visibility of our object to `false` and the `drawScene` function would stop rendering it. The integer `counter` also works as an index to the `positions` array. We retrieve a position and apply the translation to the initial model matrix to calculate our model matrix (`mat4.translate (this.modelMatrix, this.initialModelMatrix ,this.positions[this.counter]);`). We then update our counter.

```
Bullet.prototype.update=function() {

if(this.counter<this.positions.length){
mat4.translate(this.modelMatrix,this.initialModelMatrix,this.positi
ons[this.counter]);
this.counter=this.counter+1;
      }
else{
this.visible=false;
      }
mat4.scale(this.modelMatrix,this.modelMatrix,vec3.fromValues(5,5,5)
);
}
```

In the preceding code samples, we have used a constant scaling factor `5` and we have used similar constants throughout. These constant values are provided by the scene designer and we have implemented them as is in the scene.

Open `06-Bullet-Action-Complete.html` in your editor.

We first load our bullet object (`bullet.json`) in our `start` function as shown in the following code:

```
loadStageObject("model/weapons/bullet.json",
[0.0,0.0,0.0],0.0,0.0,0.0);
```

We also modified our `loadStageObject` function to create the object of our `Bullet` class. Remember that all the objects are added to the `stageObjects` array of the `Stage` class. As each object inherits `StageObject`, the `drawScene` function invokes the `update` function of each object independent of its implementation.

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
        $.getJSON(url,function(data){
...
        else if(nameOfObject==="bullet"){
stageObject=new Bullet();
bullet=stageObject;
```

```
bullet=stageObject;
            }
...
  }
}
```

We have also added a `handleKeys` function to initialize the bullet each time the Space bar is hit. The `initialize` function toggles the visibility of the bullet, and then the `drawScene` function starts processing it.

```
function handleKeys(event) {
switch(event.keyCode) {//determine the key pressed
case 32:// Space Bar
bullet.initialize();
break;
            }
        }
```

Reusing objects in multiple bullets

In this section, we would like to cover a very basic concept of reusing objects. In the preceding code, each time we hit the Space bar, the same bullet changes its location. What we would actually like is that we generate a new bullet each time the Space bar is hit. The scene should have multiple bullets. A simple strategy would have been to clone the bullet each time we hit the Space bar. However, using this way, we would have multiple objects initialized in the scene and we would have no way to track them. Hence, we initialize a pool of bullets, and then whenever we hit the Space bar, we pick a bullet from the pool based on its visibility. If the bullet is not visible, this means it is not in action and is not being rendered by the `drawScene` function. We can safely initialize an invisible bullet completely transparent to the end user.

Concepts like these actually make your game successful. This way, even if the game is being rendered for a long period of time without a pause, the game performance will not deteriorate.

Let's look at the implementation of the concept closely. Open `06-Multiple-Bullets-Action.html` in your editor.

We start with the definition of our variables, a `bullets[]`array that will hold the pool of bullets, and `nextBulletIndex` that will hold the index of the next bullet to be used.

```
var bullets=[];
var nextBulletIndex=0;
```

We just added the `loadStageObjects` and `addStageObjects` functions here. We discussed them earlier; we added them to make the flow more intuitive.

After each object is loaded in `loadStageObject`, it invokes `addStageObject` which in turn invokes `cloneObjects`. The `cloneObjects` function checks for the object type and then creates its corresponding clones by invoking the `clone` function of the object. For the bullet, it creates 24 clones (one was already loaded, in total 25) and then adds them to the `bullets` array. Each bullet also holds the reference to the `cam` object in its `camera` property.

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
...
```

```
addStageObject(stageObject,location,rotationX,rotationY,rotationZ);
...
}
function
addStageObject(stageObject,location,rotationX,rotationY,rotationZ){
cloneObjects(stageObject);
stageObject.location=location;
stageObject.rotationX=rotationX;
stageObject.rotationY=rotationY;
stageObject.rotationZ=rotationZ;
stage.addModel(stageObject);
    }

function cloneObjects(stageObject){
...
if(stageObject.name=="bullet"){
bullets.push(stageObject);
for(var j=0;j<24;++j){
var bullet=stageObject.clone();
bullet.camera=cam;
bullets.push(bullet);
stage.addModel(bullet);
            }
        }
}
```

In the `handleKeys` function, we simply pick a bullet from the array referenced by the `nextBulletIndex` variable. The value of the variable is 0 initially. After that, we increment the next bullet index, and if the `counter` has exceeded the length of the `bullets` array, then we initialize it back to 0.

```
function handleKeys(event) {
switch(event.keyCode) {//determine the key pressed
case 32://space key
bullets[nextBulletIndex].initialize();
nextBulletIndex++;
if(nextBulletIndex>=bullets.length){
nextBulletIndex=0;
                        }
break;
            }
        }
```

In the preceding code, we used a very simple algorithm to pick bullets from the pool. We only used the next index of the last active bullet. We could use simple

logic currently because we know that by using the next index, we are actually using the bullet that was either never used or was the oldest one to be used (circular) as the life of each object is the same. In other games, we might have to use more complicated logic to re-use objects because the life of each object may vary.

Using Bspline interpolation for grenade action

In this section, we will introduce the left hand. The left hand will be used to throw the grenade. So, first we will need to create a linear animation of the left hand. However, this time, we will not interpolate over position but over rotation. Then, we will chain animations. When the animation of the left hand ends, we will start the animation of the grenade.

Using linear interpolation for left-hand rotation

We want to throw the grenade when the *Enter* key is pressed. Let's first go over the class to handle the animation of the left hand.

Open the `LeftHand.js` file from the `primitive/game` folder of the code bundle in your text editor.

The `LeftHand` class also inherits `StageObject`. We have defined some new properties (`rotations`, `counter`, and `grenadeCallBack`). The `rotations` array holds the angles that we need to interpolate on. The integer `counter` maintains number of times the `update` function has been invoked. The variable `grenadeCallback`, holds the reference to the function that has to be invoked once the animation has finished. The constructor initializes the `rotations` array with values from 90 to 75 degrees. The `visible` property has been initialized to false to avoid rendering of the left arm initially:

```
LeftHand= inherit(StageObject, function (){
superc(this);
this.rotations=[];
for(var j=90.0;j>=75.0;--j){
this.rotations.push((j*22/7)/180);
```

```
    }
this.visible=false;
this.counter=0;
this.grenadeCallback=null;
});
```

In the `update` function, we obtain the camera view matrix. Calculate the camera matrix, then translate the arm with respect to the camera using the matrix. Then, apply a rotation transformation along the *x* axis with an angle from the `rotations` array. Note that in the case of the left arm, we use the latest value of the camera view matrix throughout the animation. This is done so that during the animation if you rotate or translate your camera, the arm moves with it. Also, when we have iterated over the complete set of rotations, it resets the `counter`, toggles visibility, and invokes the callback function.

```
LeftHand.prototype.update=function() {
if(this.counter<this.rotations.length) {
var mMatrix=mat4.clone(this.camera.viewMatrix);
mat4.invert(mMatrix,mMatrix);
mat4.translate(this.modelMatrix,mMatrix,vec3.fromValues(-1.5,-3,-6)
);
mat4.rotateX(this.modelMatrix,this.modelMatrix,this.rotations[this.
counter]);
this.counter=this.counter+1;

}else{
this.counter=0;
this.visible=false;
this.grenadeCallback();
    }
}
```

Open the `grenade.js` file from the `primitive/game` folder in your editor.

The `Grenade` class is very similar to the `Bullet` class except for one difference: we do not calculate the interpolates in the initialize function. We calculate it over the control points once the application is loaded in the main control code. Each time the animation is required to run, we pass the `positions` array from the main code as a parameter to the `initialize` function.

```
Grenade= inherit(StageObject, function (){
superc(this);
this.positions=[];
this.counter=0;
```

```
this.visible=false;
this.initialModelMatrix=mat4.create();
});
```

The `initialize` function of the class calculates the `initialModelMatrix` from the camera matrix, resets the `counter` and visibility, and initializes the `positions` array as shown in the following code:

```
Grenade.prototype.initialize=function(positions) {
var mMatrix=mat4.clone(this.camera.viewMatrix);
mat4.invert(this.initialModelMatrix,mMatrix);
this.counter=0;
this.visible=true;
this.positions=positions;

//mat4.translate(this.modelMatrix,mMatrix,vec3.fromValues(0,-3,-8))
;
}
```

The `update` function is exactly the same as the one in the `Bullet` class. It simply calculates the model matrix from `intialModelMatrix` and the current active position from the `positions` array on each update call from `drawScene`.

```
Grenade.prototype.update=function() {
if(this.counter<this.positions.length-1){
mat4.translate(this.modelMatrix,this.initialModelMatrix,this.positi
ons[this.counter]);
this.counter=this.counter+1;
    }
else{
this.visible=false;
    }
}
```

Open `06-Grenade-Action.html` in your editor to understand the implementation of the preceding classes.

We start with the declaration of variables to hold the objects of the grenade and our left hand.

```
var leftHand=null;
var grenade=null;
```

Then, we define the control points to calculate interpolated values for the grenade animation and also define the `grenadePositions` array to store the

interpolates. Note that these control points are in respect to the camera position.

```
var grenadeControlPoints=[[-1.5,-2,-10],[-1.5,10,-30],
[-1.5,15,-40],[-1.5,10,-60],[-1.5,0,-80]];
var grenadePositions=[];
```

The `start` function initializes the `BsplineInterpolation` object, and it calculates the interpolates and stores them in its `position` array. In the next line, we simply copy its value to the `grenadePositions` array. Then, the `start` function loads the left hand and grenade JSON models.

```
function start() {
...
var splineInterpolation=new
BSplineInterpolation(grenadeControlPoints);
grenadePositions=splineInterpolation.position;
...
loadStageObject("model/weapons/leftHand.json",
[0.0,20.0,-150.0],0.0,0.0,0.0);
loadStageObject("model/weapons/grenade.json",
[0.0,20.0,-150.0],0.0,0.0,0.0);
...
}
```

The `loadStageObject` function checks the loaded object type, and then creates the objects for `LeftHand` and `Grenade`. The `leftHand` object's property `grenadeCallback` is initialized with a reference of the `initializeGrenade` function of our main code. It is invoked once the hand animations have been executed.

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
...
else if(nameOfObject=="leftHand"){
stageObject=new LeftHand();
leftHand=stageObject;
leftHand.grenadeCallback=initializeGrenade;
            }
...
else if(nameOfObject=="grenade"){
stageObject=new Grenade();
grenade=stageObject;

            }
...
}
```

```
}
```

The `initializeGrenade` function is invoked from the `update` function of the `LeftHand` class once the animation has been executed. It invokes the `initialize` function of the `Grenade` class and passes the `grenadePositions` array calculated in the `start` function. Note that these interpolates are calculated in reference to the camera and not the world space. We calculate each position in world space by translating them with the camera matrix in the `update` function of the `Grenade` class.

```
function initializeGrenade(){
grenade.initialize(grenadePositions);
    }
```

Our `handleKeys` function simply toggles the visibility of the left hand when the *Enter* key is pressed. Once the object is marked visible, the `drawScene` function starts invoking its `update` function and subsequently leads to its rendering. Once the hand animation ends, it invokes the grenade animation from the callback function.

```
function handleKeys(event) {
...
switch(event.keyCode) {//determine the key pressed
case 13://Press Enter for grenade action
leftHand.visible=true;

break;
...
            }
        }
```

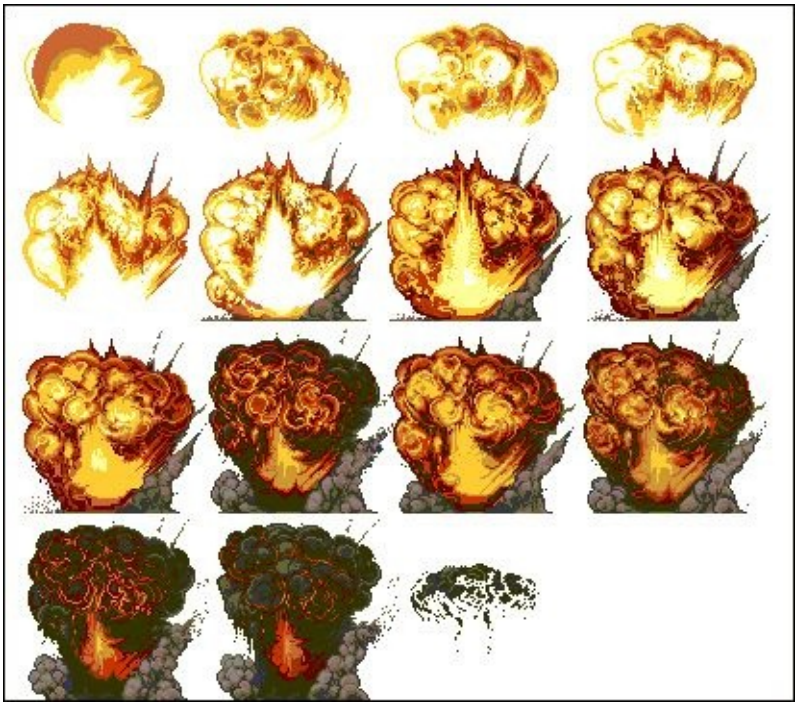
Using texture animation for an explosion effect

In the last section of this chapter, we will create an animation effect with textures. We want the grenade to explode once its animation ends. Although explosions in games are created using particle physics, for simple explosions, we chose to confine ourselves to texture animation. So let's first understand what we mean by texture animation.

Until now, we were updating an object's location or rotation in our `update` function. In short, we were manipulating the ModelView matrix. Now, we will change the texture in our `update` function.

Let's slide back to the start of the chapter where we said that the `Stage` class holds the reference to all initialized textures and each `StageObject` holds the `textureIndex`, the index to the initialized texture. Now, if we change the value of the `textureIndex` property in the update call, it will render each time with a different texture, giving an illusion of the animation.

Let's look at textures, shown in the following screenshot, which we have created to simulate an explosion:



We will apply these textures to an explosion object (`explosion.json` from the `model\weapons` folder of the code bundle).

So, first let's understand our `explosion.js` file from the `primitive/game` folder of the code bundle.

The `Explosion` class has a new `textureIndices[]`variable in the place of the `positions` array. This array holds the indices of textures defined in the primary `Stage` class.

```
Explosion= inherit(StageObject, function (){
superc(this);
this.textureIndices=[];
this.counter=0;
this.visible=false;
});
```

The `initialize` function takes the `modelMatrix` of the grenade as a parameter. The objective is simple: the explosion has to occur at the same location and orientation of the grenade in its last frame. Hence, we clone the received model matrix and simply assign it to the model matrix of the `Explosion` object. We also

initialize the `textureIndex` class variable to the first element of the `texture` array.

```
Explosion.prototype.initialize=function(modelMatrix) {
this.modelMatrix=mat4.clone(modelMatrix);
this.counter=0;
this.textureIndex=this.textureIndices[0];
this.visible=true;
this.frames=0;
}
```

The `update` function increments the `counter` and assigns the index of the next texture in the `textureIndices` array. Once the animation has finished, we toggle the visibility of the texture.

```
Explosion.prototype.update=function() {
if(this.counter<this.textureIndices.length){
this.textureIndex=this.textureIndices[this.counter];
this.counter-++;
    }
else{
this.visible=false;
this.counter=0;
    }
}
```

We also need to do a small change in the `Grenade.js` file from the `primitive/game` folder of the code bundle.

We need to add a callback function (`explosionCallBack`) to the `Grenade` class so that we can pass the model matrix of the grenade in the last frame and initialize our explosion.

```
Grenade= inherit(StageObject, function (){
...
this.explosionCallBack=null;
});
```

When the `Grenade` animation finishes, it invokes the `explosionCallBack` function and passes its `modelMatrix` as shown in the following code:

```
Grenade.prototype.update=function() {
if(this.counter<this.positions.length-1){
...
   }
```

```
else{
...
if(this.explosionCallBack!=null){
this.explosionCallBack(this.modelMatrix);
        }
    }
}
```

Let's walk through the implementation of the preceding classes in `06-Grenade-Action-Blast.html`.

We have added a new variable to hold the explosion object, as shown in the following code:

```
var explosion=null;
```

In the `loadStageObject` function, after the `grenade` object is created, we assign the name of the callback function (`initializeExplosion`) to the `explosionCallBack` property of the `grenade` object. It is invoked once its animation has finished. After the `explosion` object is created, we invoke `initExplosionTextures` to load all the textures shown previously.

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
...
else if(nameOfObject=="grenade"){
stageObject=new Grenade();
grenade=stageObject;
grenade.explosionCallBack=initializeExplosion;
}else if(nameOfObject=="explosion"){
stageObject=new Explosion();
explosion=stageObject;
initExplosionTextures();
            }
...
}
```

The textures are named `blast_1.png`, `blast_2.png`, `blast_3.png`, all the way to `blast_15.png`. We iterate over the names and load the textures. We create a unique key (`vartextureIndex=currentDate.getMilliseconds()+j-;`) to create a reference to the texture object in the `Stageclass'textures` object. Each unique key of the texture is pushed to the `explosion.textureIndices` array and the texture object is initialized and stored via the `stage.addtexture` function.

```
function initExplosionTextures(){
```

```
for(var j=1;j<=15;++j){
var fileName="blast_"+j+".png";
var currentDate=new Date();
var textureIndex=currentDate.getMilliseconds()+j*2000;
explosion.textureIndices.push(textureIndex);
textureList[fileName]=textureIndex;
var image = new Image();
image.onload = function() {
stage.addTexture(textureIndex,fileName,image); }
image.src = "model/textures/"+fileName;
        }
    }
```

In our last change in code, the `initializeExplosion` function is assigned to the `grenade` object. When the `grenade` animation finishes, it invokes this function to create a chained animation of the explosion. It also takes the `modelMatrix` of the `grenade` object as a parameter. This function in turn invokes the `initialize` function of the `explosion` object.

```
function initializeExplosion(matrix){
explosion.initialize(matrix);
    }
```

Summary

This chapter was most likely our entry point to creating games using WebGL. It covered very basic but important concepts of game development. We started this chapter with architectural updates to apply textures in our objects in the scene. We then covered the primary concept of timing of animations in game development. The basic techniques of creating animations using interpolates will be useful in creating many complex animations. Although we implemented only linear and B-spline interpolations, the core concept of using interpolation techniques was covered.

The key concept we touched upon was how simple animations have become. We plan them in reference to one object and then apply a transformation with respect to the scene. In our examples, our reference object was the first-person camera. The interpolation points that we used were constant, but when we applied the camera matrix, we produced different trajectories for the same points. This is the most important concept that we need to understand when creating animations. We finished our chapter with texture animation.

In the next chapter, we will use concepts such as collision detection to intercept the path of these animations.

Chapter 7. Physics and Terrains

In a game, we generally do not like to control our animations, typically in a case where a bullet or a grenade is involved. We want force, impulse, and gravity to control the motion in a game. This is what this chapter is about—physics simulation. We will discuss how physics engines control component trajectories as well as work with collision detection in a game. However, before we dive deep into physics, we want to extend the terrain (ground) of our game scene. We will cover the following topics in this chapter:

- A simple terrain: plane geometry
- JavaScript 3D physics engines: JigLibJS
- Adding gravity and a rigid body to the game scene
- Forces, impulse, and collision detection: grenade and bullet animation revisited
- Extending our terrain with physics

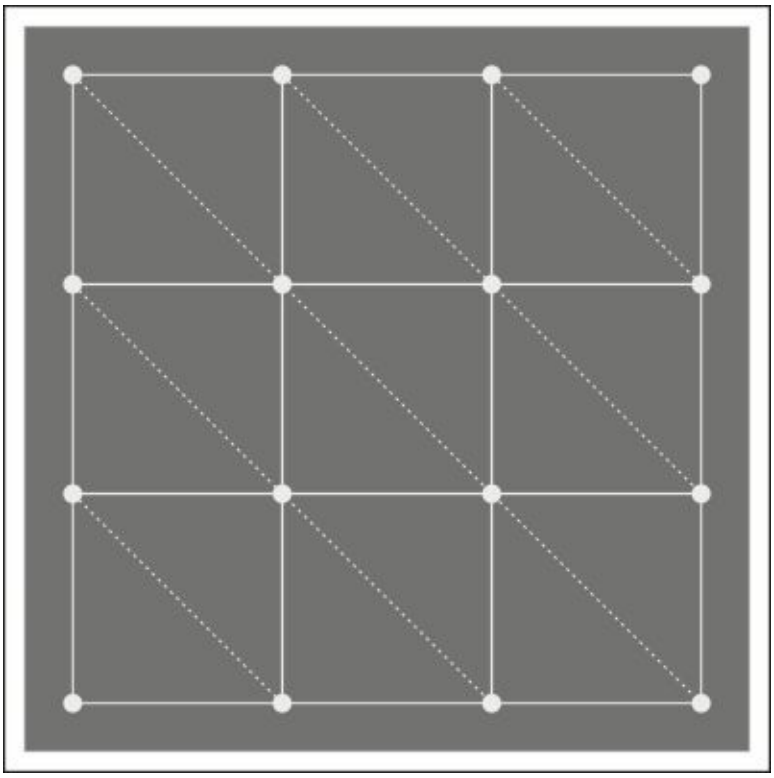
Understanding a simple terrain – plane geometry

In all our previous examples, we have used a JSON file (`model/obj/terrain.json`) as our terrain or ground base, where we have added all our objects. This approach is generally not used in games where the scene is a complete city or world. In a game, terrains are endless. The terrain should appear to meet the sky. If we clone the OBJ file multiple times to achieve the effect, then we might make the rendering slower; the same effect can be achieved with fewer polygons rendered. This is why most graphic libraries come with functionalities to create basic geometries such as a plane, sphere, octahedron, and others. In this chapter, we will focus on a plane geometry.

A **plane geometry** is mostly used to create terrains. However, we will first discuss the structure of a plane geometry. A plane geometry is made of vertices where the normal of each vertex is parallel to each other. The equation for a plane is commonly written as follows:

$$Ax + By + Cz + D = 0$$

In the following diagram, you will see that the plane geometry is made of segments. You should always have a higher number of segments. When the geometry is viewed from different camera angles, more segments lead to better resolution and minimum culling artifacts. We discussed earlier in Chapter 1, *Getting Started with WebGL Game Development*, primitive culling in the assembly/rasterization phase in the *The fragment shader* section. The primitives that fall partly outside the view frustum are culled or not passed to the fragment shader for processing. If you have a bigger segment size, then you might notice missing triangles around the corners of the terrain because the centroid of the triangle might fall outside the view area.



The preceding diagram of the plane geometry has three width segments and three height segments. To render the preceding geometry, we need to generate the following set of information:

- Vertices
- Vertex normals
- Indices to reduce vertex redundancy
- For the UV map, if we apply a texture to this geometry, we should have a texture coordinate per vertex.
- To generate vertices, we also need the width and height of the plane. Hence, to generate the information to render the geometry shown in the preceding diagram, we need the following:
    - Plane width
    - Plane height
    - Width segments
    - Height segments

Note

Terrain is one of the most important components for most games. The plane geometry computation logic is a very standard algorithm that is implemented in most WebGL libraries, and we have covered its implementation to understand the topic height maps better.

- Also, for our formulas to generate the vertex information from the preceding data, we assume that the plane is placed at the world center and lies in the XY plane ($z = 0$). The preceding diagram also shows that for n segments, we have $n + 1$ vertices on each side and we certainly want our number of vertices to be even and in the power of 2.
- Let's start with our first formula to generate vertices. Let's say that if the plane is a grid of 4 x 4 vertices and each side's width and height is 16 units and the image center is (0, 0), then the coordinates for vertices are shown in the following table:

Vertices	**Coordinates**
(0, 0)	(-32, 32, 0) These coordinates are calculated as (-32 + 64 *0/3, 32 - 64* 0/3, 0)
(1,0)	(-32 + 64/3, 32, 0)
(2, 0)	(-32 + 64 * 2/3, 32, 0)
(3, 0)	(-32 + 64 * 3/3, 32, 0)
(0, 1)	(-32 + 64 * 0/3, 32 - 64/3, 0)
(1, 1)	(-32 + 64 * 1/3, 32 - 64/3, 0)
(2, 1)	(-32 + 64 * 2/3, 32 - 64/3, 0)
(3, 1)	(-32 + 64 * 3/3, 32 - 64/3, 0)

(0, 2)	(-32 + 64 *0/3*, *32* - *64* 2/3, 0)

The rest of the vertices, (1, 2), (2, 2), (3, 2), (0, 3), (1, 3), (2, 3), and (3, 3), are calculated in a similar manner.

Let's look at our code to generate vertices. Open the `PlaneGeometry.js` file from the `primitive` folder of the code files of this chapter in your favorite editor. The following code snippet is present in this file:

```
PlaneGeometry = inherit(Geometry, function (width, height,
widthOfSegments, heightOfSegments)
{
  superc(this);
  this.width = width;
  this.height = height;
  this.widthOfSegments = widthOfSegments || 1;
  this.heightOfSegments = heightOfSegments || 1;
  var i, j;
  var widthHalf = width  2;
  var heightHalf = height  2;
  var gridX = this.widthOfSegments;
  var gridZ = this.heightOfSegments;
  var gridXN = gridX + 1;
  var gridZN = gridZ + 1;
  var segmentWidth = this.width  gridX;
  var segmentHeight = this.height  gridZ;
  var normal = vec3.fromValues( 0, 0, 1 );

  for ( i = 0; i<gridZN; i ++ ) {
    for ( j = 0; j <gridXN; j ++ ) {

      var x = j  segmentWidth - widthHalf;
      var y = i  segmentHeight - heightHalf;
      this.vertices.push(x);
      this.vertices.push(- y);
      this.vertices.push(0);

    }
  }
```

In the preceding code, first we calculate the number of vertices on each side (`gridXN = gridX + 1;` and `gridZN = gridZ + 1;`). The `segmentWidth = this.width / gridX` statement calculates the length of each segment, which in our case is 64/3. Hence, if we look at the formula in the preceding code, `var x =`

`j * segmentWidth – widthHalf`, we get the following values:

Values of i and j	**x coordinate**	**y coordinate**
For i = 0 and j = 0	x = 0 * 64/3 - 32	y = 0 * 64/3 - 32
For i = 0 and j = 1	x = 1 * 64/3 - 32	y = 0 * 64/3 - 32
For i = 0 and j = 2	x = 2 * 64/3 - 32	y = 0 * 64/3 - 32
For i = 0 and j = 3	x = 3 * 64/3 - 32	y = 0 * 64/3 - 32
For i = 1 and j = 0	x = 0 * 64/3 - 32	y = 1 * 64/3 - 32
For i = 1 and j = 1	x = 1 * 64/3 - 32	y = 1 * 64/3 - 32
For i = 1 and j = 2	x = 2 * 64/3 - 32	y = 1 * 64/3 - 32
For i = 1 and j = 3	x = 3 * 64/3 - 32	y = 1 * 64/3 - 32

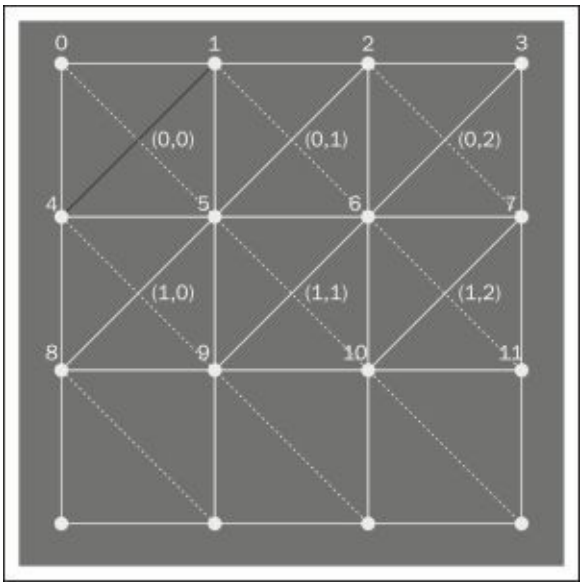
If you look at the preceding list, the value of `y` is 1 * 64/3 - 32, which is negative of the derived value that we got. Hence, we push this negative value in the code using the `this.vertices.push(- y)` statement.

Notice that we have inherited our `Geometry` class, which means we have all the variables (`this.vertices`, `this.faces`) already defined. So, we simply push the calculated vertices on our `vertices` array.

Also, as our plane lies in the XY plane, the *z* axis is our normal for each vertex. Hence, we have also defined our normal (`var normal = vec3.fromValues( 0, 0, 1 );`).

Let's now calculate the indices that make our triangles.

If you see the order in which we push the vertices on our array, you will realize that we do it in the row-major order. Hence, (0, 0), (0, 1), (0, 2), (0, 3),....., (0, *n*) are indexed as (0, 1, 2, 3,.....*n*). The following diagram shows the vertex to index mapping of the plane segments:



Hence, the indices for the square are:

- (0, 0) is (0, 4, 5, 1), the two triangles are (0, 4, 1) and (4, 5, 1)
- (0, 1) is (1, 5, 6, 2), the two triangles are (1, 5, 2) and (5, 6, 2)
- (0, 2) is (2, 6, 7, 3), the two triangles are (2, 6, 3) and (6, 7, 3)
- (1, 0) is (4, 8, 9, 5), the two triangles are (4, 8, 5) and (8, 9, 5)
- (1, 1) is (5, 9, 10, 6), the two triangles are (5, 9, 6) and (9, 10, 6)
- (1, 2) is (6, 10, 11, 7), the two triangles are (6, 10, 7) and (10, 11, 7)

For each square, the four vertices are denoted as (a, b, c, d) and their values are calculated using the segment height index, segment width index, and number of vertices, then our formula for each vertex is as follows:

- For vertex a, the formula is $j + gridXN * i$
- For vertex b, the formula is $j + gridXN * (i + 1)$
- For vertex c, the formula is $(j + 1) + gridXN * (i + 1)$
- For vertex d, the formula is $(j + 1) + gridXN * I$

Hence, the values for each vertex are shown in the following table:

Coordinates	Vertices
(0,0)	a = 0 + 4 * 0 = 0 b = 0 + 4 * (0 + 1) = 4 c = (0 + 1) + 4 * (0 + 1) = 5 d = (0 + 1) + 4 * 0 = 1 Hence, the values for the vertices are (0, 4, 5, 1)
(0,1)	a = 1 + 4 * 0 = 1 b = 1 + 4 * (0 + 1) = 5 c = (1 + 1) + 4 * (0 + 1) = 6 d = (1 + 1) + 4 * 0 = 2 Hence, the values for the vertices are (1, 5, 6, 2)
(0,2)	a = 2 + 4 * 0 = 2 b = 2 + 4 * (0 + 1) = 6 c= (2 + 1) + 4 * (0 + 1) = 7 d = (2 + 1) + 4 * 0 = 3 Hence, the values for the vertices are (2, 6, 7, 3)
(1,0)	a = 3 + 4 * 0 = 3 b = 3 + 4 * (0 + 1) = 7 c= (3 + 1) + 4 * (0 + 1) = 8 d = (3 + 1) + 4 * 0 = 4

	Hence, the values for the vertices are (3, 7, 8, 4)

The formula to calculate UV maps for the corresponding vertices is as follows:

```
var uva = vec2.fromValues( j  gridX, 1 - i  gridZ );
var uvb = vec2.fromValues( j  gridX, 1 - ( i + 1 )  gridZ );
var uvc = vec2.fromValues( ( j + 1 )  gridX, 1 - ( i + 1 )  gridZ
);
var uvd = vec2.fromValues( ( j + 1 )  gridX, 1 - i  gridZ );
```

Let's now look at the complete code to generate the faces array, UV maps, and normal array.

We first iterate over all segments of our plane. For each segment, we have two faces and add them to our `faces` array. The following code calculates the indices and UV coordinates for the four vertices of the polygon using the formula derived earlier:

```
var faceUVIndex=0;
var faceIndex=0;
this.faceVertexUvs[0]=[];
var uvs=[[]];
uvs[0]=[];
for ( i = 0; i<gridZ; i ++ ) {
for ( j = 0; j <gridX; j ++ ) {

var a = j + gridXN  i;
var b = j + gridXN  ( i + 1 );
var c = ( j + 1 ) + gridXN  ( i + 1 );
var d = ( j + 1 ) + gridXN  i;
var uva = vec2.fromValues( j  gridX, 1 - i  gridZ );
var uvb = vec2.fromValues( j  gridX, 1 - ( i + 1 )  gridZ );
var uvc = vec2.fromValues( ( j + 1 )  gridX, 1 - ( i + 1 )  gridZ
);
var uvd = vec2.fromValues( ( j + 1 )  gridX, 1 - i  gridZ );
```

The following code copies the generated UV coordinates to the `uvs` array of the `Geometry` class at the index 0. It is the index of the first material. Also our geometry will use only one material:

```
uvs[0].push(uva[0]);
uvs[0].push(uva[1]);
uvs[0].push(uvb[0]);
uvs[0].push(uvb[1]);
```

```
uvs[0].push(uvc[0]);
uvs[0].push(uvc[1]);
uvs[0].push(uvd[0]);
uvs[0].push(uvd[1]);
```

In the following code, we create a `face` object and store the calculated indices in the corresponding face properties (`a`, `b`, `d`). The normals of all vertices lie on the same plane. Hence, the normals are cloned from the normal object (0, 0, 1) and copied to the `vertexNormals` array of the `face` object:

```
var face = new Face();
face.a=a;
face.b=b;
face.c=d;
face.vertexNormals=[];
face.vertexNormals["a"]=vec3.clone(normal);
face.vertexNormals["b"]=vec3.clone(normal);
face.vertexNormals["c"]=vec3.clone( normal);
this.faces.push( face );
```

In the following code, `faceIndex` maintains the count of the number of face objects stored in the `faces` array, and we store the UV indices at the corresponding `faceIndex` element in the `faceVertexUvs` array:

```
this.faceVertexUvs[0][faceIndex]=[];
this.faceVertexUvs[0][faceIndex]["a"]= faceUVIndex;
this.faceVertexUvs[0][faceIndex]["b"]= faceUVIndex+1;
this.faceVertexUvs[0][faceIndex]["c"]= faceUVIndex+3;
faceIndex=faceIndex+1;
```

In the following code, as there are two faces per segment, we repeat the process of creating and adding the second face to the `faces` array and store their corresponding UV indices in the `faceVertexUvs` array of the geometry:

```
face = new Face();
face.a=b;
face.b=c;
face.c=d;
face.vertexNormals=[];
face.vertexNormals["a"]=vec3.clone(normal);
face.vertexNormals["b"]=vec3.clone(normal);
face.vertexNormals["c"]=vec3.clone(normal);
this.faces.push( face );
this.faceVertexUvs[0][faceIndex]=[];
this.faceVertexUvs[0][faceIndex]["a"]= faceUVIndex+1;
```

```
this.faceVertexUvs[0][faceIndex]["b"]= faceUVIndex+2;
this.faceVertexUvs[0][faceIndex]["c"]= faceUVIndex+3;
faceIndex=faceIndex+1;
faceUVIndex=faceUVIndex+4;
```

In the following code, once our `vertices` and `faces` (`a`, `b`, `c`, `vertexNormals`), `uvs`, and `faceVertexUvs` arrays of our `Geometry` class are computed, we invoke the `verticesFromFaceUvs()` function to make the number of vertices equal to the number of `uvs` (for vertex shaders). We also invoked the `indicesFromFaces()` function to copy the face attributes, `a`, `b`, and `c`, to the `indices` array of the `Geometry` class and invoke the `morphedVertexNormalsFromObj()` function to prepare our `normals` array of the `Geometry` class:

```
this.verticesFromFaceUvs(this.vertices,uvs,0);
this.indicesFromFaces();
this.morphedVertexNormalsFromObj();
```

Rendering our plane geometry

The preceding code computes our `vertices`, `indices`, `normals`, and `uvs` arrays. We generally read these values from the JSON file and store them in their corresponding arrays. So, the rendering code of the geometry is similar to any other geometry.

Open the `Plane.js` file from `primitive/game` in your favorite text editor. The objective of the code is to initialize the `PlaneGeometry` object with the required parameters (`width`, `height`, `widthOfSegments`, `heightOfSegments`). The following is the code snippet from the `Plane.js` file:

```
Plane= inherit(StageObject, function (width, height,
widthOfSegments, heightOfSegments,textureName){
  superc(this);
  this.geometry=null;
  this.width=width;
  this.height=height;
  this.ws=widthOfSegments;
  this.wh=heightOfSegments;
  this.geometry=new PlaneGeometry(width, height, widthOfSegments,
heightOfSegments;
  this.materialFile=textureName;//"terrain_1024.jpg";
});
```

We also pass the name of the texture to be associated with the UV map of the geometry.

Open the `07-New-Terrain.html` file in your favorite editor. In this code, we simply initialize our `PlaneGeometry` object and set the corresponding material file name:

```
function start() {
...
  var plane=new StageObject();
  plane.geometry=new PlaneGeometry(7500,7500,63,63,null);

  plane.materialFile="terrain_1024.jpg";
  loadTexture(plane,clampToEdge);
...
}
```

The function, `loadTexture()`, is another refactoring attempt. We have moved

the code to load textures from the `loadStageObject` function. This function is now invoked from the `start()` function as you saw in the preceding code and also from the `loadStageObject()` function:

```
function loadTexture(stageObject,clampToEdge){
  if(stageObject.materialFile!=null){
    if((textureList[stageObject.materialFile.trim()]===undefined)){
      var currentDate=new Date();
      var textureIndex=currentDate.getMilliseconds();
      stageObject.textureIndex=textureIndex;
      textureList[stageObject.materialFile.trim()]=textureIndex;

initTextures(stageObject.materialFile.trim(),textureIndex,bool);
    }
    else{

stageObject.textureIndex=textureList[stageObject.materialFile.trim(
)]
    }
  }
}
```

We have also modified our `Stage.js` file's code located in the `primitive` folder to accommodate a new parameter, `clampToEdge`. It is a Boolean value to decide whether the wrapping mode (`this.gl.CLAMP_TO_EDGE`) has to be set for this texture if the UV coordinates fall outside the range of 0 to 1. For terrains, we want to make sure that if the UV map exceeds the range, then the texture is not repeated and it covers the geometry. The texture parameters have been explained in Chapter 4, *Applying Textures*.

Open the `Stage.js` file from the `primitive` folder in your favorite editor. The following code is present in this file:

```
addTexture:function(index,name,img,clampToEdge){
...
  if(clampToEdge){
    this.gl.texParameteri(this.gl.TEXTURE_2D,
this.gl.TEXTURE_WRAP_S, this.gl.CLAMP_TO_EDGE);
    this.gl.texParameteri(this.gl.TEXTURE_2D,
this.gl.TEXTURE_WRAP_T, this.gl.CLAMP_TO_EDGE);

  }
...
}
```

We will now understand JavaScript physics engines and cover terrain physics in the last section of the chapter.

Comparing JavaScript 3D physics engines

Let's start by looking at the most popular JavaScript physics engines. Most of them are still under development. The three most popular JavaScript physics libraries are Box2dweb, Ammo.js, and JigLibJS. We will give you a quick introduction to each one and then use JigLibJS in our code.

Ammo.js

Ammo.js is directly ported from the Bullet physics engine (a C++ physics library) using Emscripten (https://github.com/kripken/emscripten). There is no human rewriting involved in the translation of the source code. The full form of **Ammo** is **Avoided making my own**. Ammo.js is a fully featured, rich physics library with a wide range of options for physics shapes. It supports all features such as collision detection, constraints, and vehicle systems.

The overview of Ammo.js is as follows:

- **Performance**: As Ammo.js is a direct port, the JavaScript code has not been optimized to run in the browser. It is powerful but performance can be an issue.
- **Features**: It is one of the most complete physics libraries available in any programming language. The Bullet physics engine has been used in many games such as *Grand Theft Auto IV* and *Red Dead Redemption*, and movies such as *2012* and *Sherlock Holmes*.
- **Usability**: The API can be confusing at times and the autogenerated documentation isn't of much help.
- **Overall**: If you are looking for a fully featured physics library, then Ammo.js will get the job done. It takes some time to become familiar with the parts of the library, but you can quickly develop complex scenes with rich interactions.

Box2dweb

Box2dweb is a port of the famous 2D physics engine Box2D. Box2D is originally written in C++. As the name suggests, you can only play with it in two dimensions. It has the most powerful constraint system and exists as **joints** classes. It has easy-to-configure options for complete scenes and individual objects.

The overview of Box2dweb is as follows:

- **Performance**: Box2dweb is very fast unless it is configured incorrectly or the graphics is heavy. The computations involved in a 2D engine are less compared to a 3D engine.
- **Features**: Box2D was not meant to do 3D. If you only need two dimensions, this library should have everything you need.
- **Browser support**: Chrome, FireFox, IE9, Safari, and Opera.

JigLibJS

JigLibJS is a port of the powerful physics library, JigLib (https://github.com/supereggbert/JigLibJS), and it was written originally in C++. The port is completely handwritten in JavaScript and is very well optimized to run in a browser. However, JigLibJS is not as feature-rich as Ammo.js or Box2Dweb. It offers three types of basic constraints:

- **Point**: This constraint helps in joining two objects.
- **WordPoint**: This constraint explains that an object is fixed to a position in the world.
- **MaxDistance**: This constraint limits the distance between two rigid bodies.

The other libraries offer very powerful constraint systems.

The overview of JigLibJS is as follows:

- **Performance**: It is customized and tuned for JavaScript. Hence, this gives high performance.
- **Features**: Although JigLibJS lacks some of the less common features that Ammo.js has, it still has enough to cover almost everything.
- **Overall**: We believe JigLibJS is able to fit most developer's needs without requiring users to have more powerful computers.
- **Browser support**: Chrome, Firefox, IE9, Safari, and Opera.

Note

We chose JigLibJS for this book as it is a handwritten port. We can read the code to understand and implement most concepts and even extend its functionalities as the library is well-structured.

Comprehending the physics engine concepts

Broadly speaking, a physics engine provides three aspects of functionality:

- Moving items around according to a set of physical rules
- Checking for collisions between items
- Reacting to collisions between items

Physics can be applied to any element of a game scene such as characters, terrains, and particles. A physics engine for games is based around Newtonian mechanics, that is, the three simple rules of motion that we learned at school. These rules are used to construct differential equations, thereby describing the movement of our simulated items. The differential equations are then solved iteratively by the algorithms that are used in the libraries. This results in believable movements and interaction of the scene elements. Within the context of a game, the physics engine provides the motion of all of the elements of the game world; therefore, it needs to be able to move any item in the world.

The physics engine for a game tends to be a separate entity which links to the rest of the code through an interface. It does not care about the entities it is moving, it just cares about their physical size, weight, velocity, and other such properties. It is a module that is distinct to the game code, renderer code, audio code, and so on.

Updating the simulation loop

A physics system typically operates by looping through every object in the game, updating the physical properties of each object (position, velocity, and so on), checking for collisions, and reacting to any collisions that are detected.

The simulation loop, which calls the physics engine for each object of interest, is kept as separate as possible from the rendering loop, which draws the graphical representation of the objects. Indeed the main loop of our game consists of just two lines of code: one calls the render update and the other line calls the simulation update. The simulation should be able to run without rendering anything to screen if the rendering and simulation aspects of the code are correctly decoupled. The part of the code which accesses the physics engine will typically run at a higher frame rate than the rest of the game, especially the renderer.

Let's now first use our JigLibJS to initialize our physics system and also plan when we want to update our physics objects. The following code snippet contains the `jigLib.PhysicsSystem.getInstance()` method:

```
var system = jigLib.PhysicsSystem.getInstance();
system.setCollisionSystem(); // CollisionSystemBrute
// system.setCollisionSystem(true); // CollisionSystemGrid
system.setSolverType("FAST");
// system.setSolverType("NORMAL");
//system.setSolverType("ACCUMULATED");
```

We first initialize our physics system (`jigLib.PhysicsSystem.getInstance()`). Then, we set the type of collision system (brute/grid).

Brute force would store two lists of collision objects in a collision system. One list would be for dynamic objects and the other would include both dynamic and static objects (each dynamic object would be in both lists). For each frame, it would loop through the dynamic list and pass each object to the larger list and check for collisions.

Grid collision is a system that uses spatial hashing, in which we basically establish a grid. In a grid, we mark down what is in touch with each grid. Then, we later go through the relevant cells of the grid and check whether everything in each relevant cell is actually intersecting with anything else in the cell. The grid collision is related to the concept of space partition.

Physics engines are split into two major parts: collision detection and collision resolution. The solver is just responsible for the latter.

After your collision detection part determines which pairs of objects collide and where they collide, the solver is responsible for creating the correct physical response. Basically, the solver type defines the differential equations to be used. `ACCUMULATED` is the most accurate and slowest differential equation.

Once we have initialized our engine, we invoke it at every render. The following function checks and updates the object's properties:

```
system.integrate(elapsedTime / 1000 );
```

Clearly, if there are large numbers of game entities that require physical simulation, this can become computationally expensive. We try to reduce the number of objects simulated by the physics engine at any time. We can use techniques similar to the ones used in the graphics code to limit the number of objects submitted to the renderer. However, culling objects based on the viewing frustum is not a good way of deciding which objects receive a physics update. If objects' physical properties cease to be updated as soon as they are off camera, then no moving objects would enter the scene. Any objects leaving the scene would just pile up immediately outside the viewing frustum, which would look terribly messy when the camera pans around.

A more satisfactory approach is to update any object that may have an interaction with the player or which the player is likely to see on the screen, either now or in the near future.

Hence, we add and remove objects from the graphics system on different events in order to improve the speed of our game. We can add and remove objects with the following lines of code:

```
system.addBody(plane.rigidBody); //To add an object
system.removeBody(plane.rigidBody);//To remove an object
```

Learning about objects in the physics system

Each item simulated by the physics engine requires some data representing the physical state of the item. This data consists of parameters that describe the item's position, orientation, and movement. Depending on the complexity and accuracy of the physical simulation required, the data and the way in which it is used becomes more detailed. The simpler the physical representation is, the cheaper the computational cost is, and therefore, the greater the number of items that can be simulated. The three main types of simulation are particles, rigid bodies, and soft bodies.

Particles

The simplest representation of physical properties is achieved through the particles method. In this case, it is assumed by the physics engine that items consist of a single point in space (that is, a particle). The particle can move in space (that is, it has velocity), but it neither rotates nor does it have any volume. We use particles in games for effects such as explosions and smoke.

Rigid bodies

The most common physical representation system is that of rigid bodies. In this case, items are defined by the physics engine as consisting of a shape in space (for example, a cube or a collection of spheres). The rigid body can move in space and can rotate in space, so it has both linear and angular velocity. It also has volume. This volume is represented by a fixed shape which does not change over time, hence the term **rigid body**. This approach is taken for practically everything in games where reasonable accuracy is required.

The rigid bodies that do not move are static bodies and the other are dynamic bodies. To define a rigid body as static, we use the `set_movable(false)` function. Terrains are the best examples of static rigid bodies.

JigLib, like other physics systems, has the `RigidBody` class. We are listing the most common properties of the class:

```
RigidBody.prototype._id=null;
RigidBody.prototype._skin=null;
RigidBody.prototype._type=null;
RigidBody.prototype._boundingSphere=null;
RigidBody.prototype._boundingBox=null;
RigidBody.prototype._currState=null;
```

```
RigidBody.prototype._currState=null;
RigidBody.prototype._oldState=null;
RigidBody.prototype._mass=null;
RigidBody.prototype._bodyInertia=null;
RigidBody.prototype._force=null;
RigidBody.prototype._torque=null;
RigidBody.prototype._activity=null;
RigidBody.prototype._movable=null;
RigidBody.prototype._doShockProcessing=null;
RigidBody.prototype._material=null;
RigidBody.prototype._nonCollidables=null;
RigidBody.prototype._constraints=null;
RigidBody.prototype.collisions=null;
RigidBody.prototype.isActive=null;
```

The `_skin` property is used to hold the transformations of the graphic element to which we want to attach the rigid body. The `_currState` property has two components: `_currState.position` and *`currState.get`*`orientation`. The position vector in JigLib is defined by an array of the three elements. JigLib has its own JVector3DUtil to manipulate the 3D vectors. The *`currState.get`*`orientation().glmatrix` statement returns an object of type mat3 (an older version of glMatrix). The collisions objects hold the collisions data.

The `RigidBody` class does not define the shape of the object in itself. The properties that hold the shape of the object are `_boundingBox` and `_boundingSphere`. The `_boundingBox` property is of the type `JAABox`. The `JAABox` class has functionalities to handle collision points defined by shapes such as box, sphere, and capsule.

Soft bodies

Items that need to change shapes are often represented in the physics engine as soft bodies. A soft body simulates all the aspects of a rigid body (linear and angular velocity as well as volume) but with the additional feature of a changeable shape, that is, deformation. This approach is used for items such as clothing, hair, and wobbly alien jellyfish. It is considerably more expensive, both computationally and memory-wise, than the rigid body representation.

Understanding the physics shapes

We have already discussed how the physics simulation should be decoupled as much as possible from the rendering loop. This also applies to the data structures and to the shapes and meshes of the game objects. The object which is rendered onto the screen can be of any shape and can be made up of many polygons. However, it is not practical to simulate large numbers of complexly shaped objects in the physics engine.

Almost every object that is simulated by the physics engine will be represented as a simple convex shape, such as a sphere or cuboid, or as a collection of such shapes. Calculating collisions and penetrations between objects can be a very expensive process. So simplifying the shapes that represent the simulated objects greatly improves the required computation. Hence, you will see that we use very simple objects and shapes to represent our 3D models in physics simulations.

JigLib has defined some useful physics shapes which we generally attach to our graphic geometries. `JBox`, `JSphere`, `JCapsule`, `JPlain`, `JTerrain`, and `JTriangleMesh` are the most commonly used objects that inherit the `RigidBody` class. They have some extra properties that define the shape, for example, `JBox` has width, height, depth, and `JSphere` has radius.

The `JTerrain` object takes the `ITerrain` interface's object as a parameter. The `ITerrain` interface's object holds data such as height, width, width segments, and height segments. We will discuss this object in detail in the section *Extending our terrain with physics*.

The `JTriangleMesh` object is another exciting object of the rigid body. It has the `createMesh(this.vertices, this.indices)` function that creates any geometry in the physics system. The vertices and indices define the shape of the 3D geometry. They are absolutely similar to our vertices and indices data. Each element of the `vertices` array represents the vertex (*x*, *y*, *z*). Each triangle [a, b, c] of the mesh is stored as an element of the `indices` array. This object is computationally expensive to use as we have to check each polygon of the mesh for collisions while in other cases, we use a bounding box. We generally avoid using it.

Adding gravity and a rigid body to the game scene

Well, it is time to add some physics to our game scene. In the example, we will simply load a JSON sphere and add it to the scene. The 3D object's motion will be controlled by the physics engine and gravity.

Open the `Sphere.js` file from `primitive/game` in your favorite editor. The sphere object inherits the `StageObject` class and has a new function, `initializePhysics`; we have also overridden the `update` function. The `initializePhysics` function is given in the following code snippet:

```
Sphere= inherit(StageObject, function (){
  superc(this);
  this.visible=false;
});
Sphere.prototype.initializePhysics=function(){
  var sphere = new jigLib.JSphere(null, 20);
  sphere.set_mass(50);
  this.rigidBody=sphere;
  this.rigidBody.moveTo(jigLib.Vector3DUtil.create(0,100,120));
  this.system.addBody(this.rigidBody);
}
```

The `initializePhysics` function initializes a physics shape, a sphere, by creating a `jigLib.JSphere` object. The object takes two parameters: the skin object and the radius of the sphere. The skin object is a reference to the object that holds the orientation of the geometric shape, which in our case is a sphere. We are not using a skin parameter in our code. We just pass the radius (`20`) to set up collision points. Then, we set the mass of our object so that gravity can accelerate the falling mass. Then, we set the location of the object in reference to the world space using the `rigidBody.moveTo` function. The location of the object in the physics world in JigLib is defined by an array of [*x*, *y*, *z*] values, and the `jigLib.Vector3DUtil.create(0,100,120)` function returns an array [0,100,120]. We then add the object to the physics system (`this.system.addBody(this.rigidBody);`). Note that unless the object is added to the physics system, the objects properties will not change. The class variables, `rigidBody` and `system`, are inherited from the `StageObject` class. The system variable holds the reference to `jigLib.PhysicsSystem` initialized in the

main control code.

The following code connects to the physics system with the geometry—the last missing piece:

```
Sphere.prototype.update=function() {
  if(this.rigidBody){
    var pos = this.rigidBody.get_currentState().position;
    this.location=vec3.fromValues(pos[0], pos[1], pos[2]);
    mat4.identity(this.modelMatrix);
    mat4.translate(this.modelMatrix,this.modelMatrix,
this.location);

mat4.scale(this.modelMatrix,this.modelMatrix,vec3.fromValues(10,10,
10));
  }   else{
    mat4.identity(this.modelMatrix);
    mat4.translate(this.modelMatrix,this.modelMatrix,
this.location);
    mat4.rotateX(this.modelMatrix,this.modelMatrix,this.rotationX);
    mat4.rotateY(this.modelMatrix,this.modelMatrix,this.rotationY);
    mat4.rotateZ(this.modelMatrix,this.modelMatrix,this.rotationZ);

mat4.scale(this.modelMatrix,this.modelMatrix,vec3.fromValues(10,10,
10));
  }
}
```

We first check whether an active `rigidBody` variable is assigned to the geometry. If it is assigned, then we retrieve the position of the `rigidBody` variable using the `this.rigidBody.get_currentState().position` function. We convert the position of `rigidBody` to the `vec3` object and set the `location` property of the sphere. The `modelMatrix` parameter of the sphere is initialized to the identity matrix and then the sphere's `modelMatrix` is translated by the `rigidBody` variable's position.

So basically, we change the location of the sphere to the `rigidBody` variable's location, and if `rigidBody` for that object does not exist, then we use the object's location and rotational values.

Open the `07-Simple-Physics-Sphere-Falling.html` file in your editor.

We initialize our physics engine in the `init_jiglib()` function. This function is

invoked when the graphics library is initialized in the `start()` function:

```
function start() {
...
  init_jiglib();
...
}
function init_jiglib() {
  system = jigLib.PhysicsSystem.getInstance();
  system.setCollisionSystem(); // CollisionSystemBrute
  system.setSolverType("FAST");
  system.setGravity(jigLib.Vector3DUtil.create( 0, -9.8, 0, 0 ) );
  var ground = new jigLib.JPlane();
  ground.set_y(0);
  ground.set_rotationX(90);
  ground.set_movable(false);
  system.addBody(ground);
}
```

In the `init_jiglib()` function, we first initialize our physics engine and then initialize our collision system using the brute force algorithm (each object is checked for collision) and set the solver type to `FAST`. Then, we initialize gravity in our system (`system.setGravity(jigLib.Vector3DUtil.create( 0, -9.8, 0, 0 ) );`). As the physics system follows the upward direction along the *y* axis, we set our gravity to a negative value.

Then, we initialize our first physics rigid body. We add a plane to our physics system. We instantiate the `jigLib.Plane` object, and set its location of `y` to `0`. The important thing to understand is that the `jigLib.Plane` object by default is aligned to the XY plane. However, we want to align it to the XZ plane. Hence, we rotate our plane by 90 degrees around the *x* axis. The `jigLib.Plane` rigid body has many applications but we want to use it as a terrain. Hence, we set it as a static body using the `RigidBody` class's function, `set_movable(false)`, and then we add it to our physics system.

The `animate()` function is explained in the following code:

```
function animate(){
...
  var updateIterations = Math.floor(elapsedTime
MINIMUM_FRAME_RATE);
  system.integrate(elapsedTime  1000 );
  while(updateIterations> 0){
    drawScene();
```

```
    updateIterations -= 1;
  }
...
}
```

We invoke the `system.integrate()` function from our `animate()` function along with the `drawScene()` function call. Basically, the `system.integrate()` function is our simulation update. It updates our physics system variables with the elapsed time (time-based update). The `loadStageObject` function is as follows:

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
...
  else if(nameOfObject=="Sphere"){
    stageObject=new Sphere();
    stageObject.system=system;
    stageObject.initializePhysics();
    stageObject.visible=true;
    sphere=stageObject;
  }
...
}
```

Finally, we load our sphere's JSON object and instantiate our `Sphere` class. We assign the physics system variable to the sphere's class variable before initializing the rigid body in the `initializePhysics` call.

When we view the result in the browser, the sphere will fall on the terrain and bounce on it. The bounce can be controlled by the restitution factor (cor) which is a property defined in the `RigidBody` class. The complete visual effect is because we update the position of our sphere geometry along the position of the rigid body.

If you notice, the physics update loop is completely decoupled from the render update. The physics simulation `update` method modifies all rigid body properties. It detects and responds to collision, like the sphere's properties in our case, such as position, orientation, and velocity, are updated after every `system.integrate()` function call. In our render call, we update our geometries' location and orientation using the rigid bodies' properties.

Implementing forces, impulse, and collision detection

JigLibJS offers functions to add force and apply impulse to a rigid body. Let's walk through the list of available functions and how they work:

```
addBodyForce=function(f, p)
f: [x,y,z] magnitude of force along three axes.
p:[x,y,z] position of force
addWorldForce=function(f, p)
f: [x,y,z] magnitude of force along three axes.
p:[x,y,z] position of force
addWorldTorque=function(t)
t: [x,y,z] magnitude of torque along three axes.
addBodyTorque=function(t)
t: [x,y,z] magnitude of torque along three axes.
applyWorldImpulse=function(impulse, pos)
impulse: [x,y,z] magnitude of impulse(mass*velocity) along three
axes.
pos:[x,y,z] position of force
applyBodyWorldImpulse=function(impulse, delta)
impulse: [x,y,z] magnitude of impulse(mass*velocity) along three
axes.
pos:[x,y,z] position of force
```

We can apply "body force", "world force", "body torque", and "world torque" in JigLibJS. The only difference between applying the `addBodyForce` and `addWorldForce` functions is the coordinate system used for the force direction. The `addBodyForce` function applies the force based on the body's coordinates. So, if we apply a force in the forward direction, the force is directly forward from the orientation that the body is currently facing. If you want absolute coordinates, you should apply the force using the `addWorldForce` function. The force only affects the body that you apply it to but the coordinate system is that of the world. The direction of the force is independent of the direction in which the body is facing. The same is true for torques.

When applying forces, we also have the option to apply the force on any point on the object other than the center of mass. This is simply given as a position vector of the location, where the force is being applied from its center of mass. Torques, however, are always applied around the center of mass. To create a more complex torque, such as a wrench (force at one end and a connection at the

other), you would need to apply a body force to a position on the end of the wrench with the other end attached to a constrained joint.

Impulses are instant changes in the velocity (or angular velocity) of the body. Rather than applying a force that accelerates a body from its initial velocity, applying an impulse immediately increases to the new velocity. This would be same as adding impulse/mass to the velocity of the body. Applying a "negative impulse" would be subtracting impulse/mass from the current velocity.

Diving deep into collision detection

The `RigidBody` class has a `collisions` array. On every rigid body collision, a `CollisionInfo` object is added to the array. The `CollisionInfo` object has the following properties:

```
CollisionInfo.prototype.objInfo=null;
CollisionInfo.prototype.dirToBody=null;
CollisionInfo.prototype.pointInfo=null;
CollisionInfo.prototype.satisfied=null;
```

We would like to limit our discussion to the `objInfo` property. The `objInfo` property is of the `CollDetectInfo` type. The `CollDetectInfo` object has the following properties:

```
CollDetectInfo.prototype.body0=null;
CollDetectInfo.prototype.body1=null;
```

The variables, `body0` and `body1`, are of the `RigidBody` type. Hence, we can access information about both objects from the `objInfo` object. The sample code to access the collision information is as follows:

```
var collidingBody=this.rigidBody.collisions[0].objInfo.body1;
```

JigLib also offers events to detect collisions. The `JCollisionEvent`=function(body, impulse) is raised when a collision occurs.

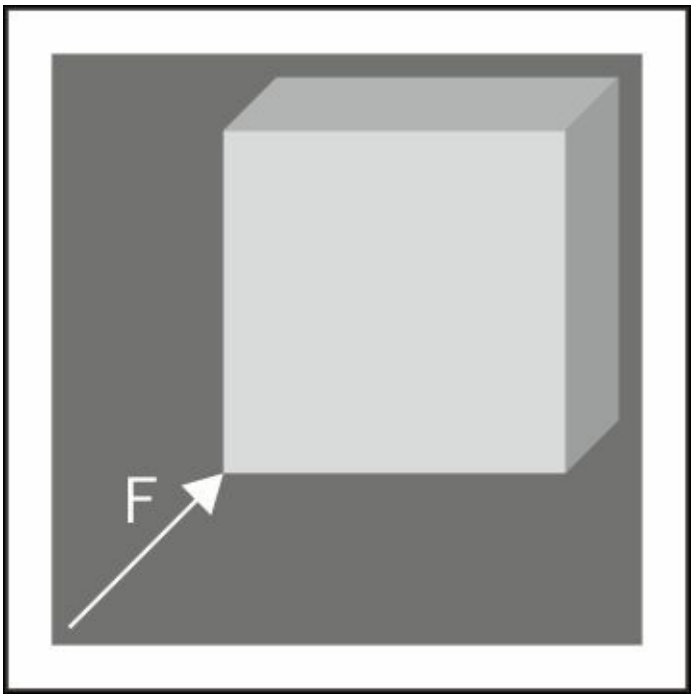
The following code attaches an event handler to a rigid body and simply logs the event info. The `body` object holds the information of the other rigid body it collided with:

```
this.rigidBody.addEventListener(jigLib.JCollisionEvent.COLLISION,fu
nction(event){
  console.log(event);
  //event.collisionBody
  //event.collisionInfo
});
```

Revisiting the grenade and bullet actions

In this code packet, we will demonstrate how the grenade action should actually be implemented. In the earlier code packet, the grenade motion was controlled by a set of defined points transformed with the camera matrix. The solution is not perfect because the length of the animation is limited and does not simulate the real-world effect in instances such as when the object path was disrupted by another object.

So, let's first revisit our grenade code. Open the `Grenade.js` file from `primitive/game` in your favorite editor. We will start with the initialization of the rigid body (sphere) in the `Grenade` class and then apply an impulse in a specific direction, as expressed in the following diagram:



The preceding diagram shows an impulse/force applied on the *y* axis and in the -Z direction at the centroid of the cube's rigid body.

In our grenade `initialize` function, we initialize our rigid body physics. We invoke the `this.initializePhysics()` function after we calculate our camera matrix. We still need our camera matrix for two things: the initial location of the

grenade and the final orientation of the force/impulse applied on the grenade. Remember, we are using the first-person camera, and all initial values (location, force) will be with respect to our model (camera) space. Then, we will transform it to world space using the model matrix. Our grenade `initialize` function is defined as follows:

```
Grenade.prototype.initialize=function(positions) {
  var mMatrix=mat4.clone(this.camera.viewMatrix);
  mat4.invert(this.initialModelMatrix,mMatrix);
  this.initializePhysics();
  this.counter=0;
  this.visible=true;
}
```

In the upcoming `initializePhysics` function's code, we first use a simple geometry, `jigLib.JSphere(null, 10)`, to present our grenade in the physics world with a radius of `10`. Then, we set its mass to `10`. The important thing is that the moment we add the sphere to the physics world, a force of 98 will be applied on it from the -Y direction, because of gravity (9.8), with instantaneous velocity of 0. We position our `rigidBody` variable in the world using the coordinate, `[-1.5,-2,-10]`, with respect to the camera and convert it to world coordinates by multiplying it by the `initialModelMatrix` (`jigLib.GLMatrix.multiplyVec3(this.initialModelMatrix,[-1.5,-2,-10]);`). We then move our rigid body to the calculated location (`this.rigidBody.moveTo(newPos[0], newPos[1], newPos[2]);`).

Now, we want to apply a force or impulse. Before we do that, let's evaluate our situation:

- We want the grenade to move along the +Y and -Z directions, so the force has to be greater than the -Y force of gravity.
- We want the grenade to move in the direction of the camera. The `initialModelMatrix` has two components: orientation and position. We want the magnitude and direction of the force to be affected only by the orientation and not the position of the camera.
- We want to apply the force on the center of the rigid body.

To tackle the first point, we defined the force by the vector [0, 300, -1200]; 300 will counter -98 of gravity, and -1200 will move the grenade in the -Z direction with respect to the camera space.

The second point requires some understanding of the camera matrix. In Chapter 5, *Camera and User Interaction*, we described the structure of a camera matrix:

Right.x	Up.x	Look.x	0.0f
Right.y	Up.y	Look.y	0.0f
Right.z	Up.z	Look.z	0.0f
Positon.x	Positon.y	Positon.z	1.0f

Hence, if we want to remove the position element from the camera matrix, we can set *m30*(12), *m31*(13), and *m32*(14) to 0. This is what the `extractOrientation` function does. It removes the position component from the camera matrix.

Now to get the final force applied with the correct direction, we transform the force vector with the orientation matrix (`initialModelMatrix` with the `0` position, `this.extractOrientation(this.initialModelMatrix); var force= jigLib.GLMatrix.multiplyVec3(this.initialModelMatrix, [0,300,-1200]);`).

The position of the force is the location of the rigid body in the world space (`pos`), and the function `addWordlForce` of the `rigidBody` class is used to apply force (`this.rigidBody.addWorldForce (force,jigLib.Vector3DUtil.create (pos.x,pos.y,pos.z));`). The `initializePhysics` function is shown in the following code:

```
Grenade.prototype.initializePhysics=function(){
  var sphere = new jigLib.JSphere(null, 10);
  sphere.set_mass(10);
  this.rigidBody=sphere;
  var newPos=jigLib.GLMatrix.multiplyVec3(this.initialModelMatrix,
[-1.5,-2,-10]);

this.rigidBody.moveTo(jigLib.Vector3DUtil.create(newPos[0],newPos[1
],newPos[2]));
  var pos = this.rigidBody.get_currentState().position;
  this.extractOrientation(this.initialModelMatrix);
  // var
force=jigLib.GLMatrix.multiplyVec3(this.initialModelMatrix,
[0,300,-1200]);
  var impulse=jigLib.GLMatrix.multiplyVec3(this.initialModelMatrix,
[0,180,-800]);
```

```
//this.rigidBody.addWorldForce(jigLib.Vector3DUtil.create(force[0],
force[1],force[2]),jigLib.Vector3DUtil.create(pos.x,pos.y,pos.z));

this.rigidBody.applyWorldImpulse(jigLib.Vector3DUtil.create(impulse
[0],impulse[1],impulse[2]),jigLib.Vector3DUtil.create(pos.x,pos.y,p
os.z));
  this.system.addBody(this.rigidBody);
  this.rigidBody.addEventListener(jigLib.JCollisionEvent.COLLISION,
function(event){
    console.log(event);
    //event.collisionBody
    //event.collisionInfo
  });
}
```

In the preceding code, we showed both the impulse and the force application code examples. However, we have commented out the force code because for the grenade, we can use either force or impulse, but actually we need to give it just some initial velocity in the desired direction. Also, if you notice, we have added code to handle the collision event but just logged the event (`this.rigidBody.addEventListener (jigLib.JCollisionEvent.COLLISION,function(event)`). For this code, we chose to use the `collisions` array, which is explained later. The `extractOrientation` function is shown in the following code:

```
Grenade.prototype.extractOrientation=function(matrix){
  matrix[12]=0;
  matrix[13]=0;
  matrix[14]=0;
}
```

Our `update` function invoked from the `drawScene` function is straightforward. It simply reads the position of the rigid body and translates the geometric shape of the grenade's `modelMatrix` with it.

We also check for the `collisions` array in our `update` function. If the length of the `collisions` array is greater than `0`, this denotes that a collision has occurred. We toggle the visibility of the grenade and remove it from the physics world (`this.system.removeBody(this.rigidBody);`). Remember, we need to remove objects that are not visible to the user from the physics world; otherwise, you will slow down the system. We then invoke the event handler (`this.callBack (this.rigidBody, this.rigidBody.collisions);`). The variable,

`this.callback`, is declared in the parent class `StageObject`. We also clear our `collisions` array (`this.rigidBody.collisions=[];`). Our `update` function is as follows:

```
Grenade.prototype.update=function() {
  if(this.rigidBody){
  var pos = this.rigidBody.get_currentState().position;
  mat4.identity(this.modelMatrix);
  this.location=vec3.fromValues(pos[0], pos[1], pos[2]);
  mat4.translate(this.modelMatrix,this.modelMatrix, this.location);

mat4.scale(this.modelMatrix,this.modelMatrix,vec3.fromValues(5,5,5)
);
  if(this.rigidBody.collisions.length>0){
    this.system.removeBody(this.rigidBody);
    console.log(this.rigidBody.collisions);
    if(this.callBack){
      this.callBack(this.rigidBody,this.rigidBody.collisions);
    }
    this.visible=false;
    this.rigidBody.collisions=[];
  }
}
```

Now, let's quickly look at the implementation of the preceding code. Open the `07-Grenade-Action-Blast-Physics.html` file in your editor. The following code snippet is present in this file:

```
function
loadStageObject(url,location,rotationX,rotationY,rotationZ){
...
  else if(nameOfObject=="grenade"){
    stageObject=new Grenade();
    grenade=stageObject;
    grenade.system=system;
    grenade.callBack=initializeExplosion;

  }
...
}
```

In our preceding code, we initialize our `grenade` object and assign the physics object and the collision `callback` function. The collision `callBack` function takes `rigidBody` as a parameter. We retrieve the position of `rigidBody`, create a `modelMatrix` parameter, and translate the model matrix to the `rigidBody`

parameter's position. We then initialize our explosion with `modelMatrix`. The `initializeExplosion` function is given in the following code:

```
function initializeExplosion(rigidBody,collisions){
  var pos = rigidBody.get_currentState().position;
  var modelMatrix=mat4.create();
  mat4.identity(modelMatrix);
  var location=vec3.fromValues(pos[0], pos[1], pos[2]);
  mat4.translate(modelMatrix,modelMatrix,location);
  explosion.initialize(modelMatrix);
}
```

Cheating in the bullet action

In this section, we will talk about cheating in the physics world. For simplicity, the code of the bullet should be similar to the grenade because all we need to do is to change the direction vector of the impulse/force. The following code demonstrates the application of impulse on our bullet object:

```
var impulse=jigLib.GLMatrix.multiplyVec3(this.initialModelMatrix,
[0,0,-2000]);
this.rigidBody.applyWorldImpulse(jigLib.Vector3DUtil.create(impulse
[0],impulse[1],impulse[2]),jigLib.Vector3DUtil.create(pos.x,pos.y,p
os.z));
```

We needed a strong impulse in the -Z direction, and that is it. However, most of the time in games, we do want physics collisions detection but do not want the physics engine to control the trajectory. So, let's visit a similar case. Open the `Bullet.js` file from `primitive/game` in your text editor.

The `initialize` function of the `Bullet` class is absolutely the same, except we check if the rigid body is initialized. We initialize its position and add the body to the physics system as follows:

```
Bullet.prototype.initialize=function() {
...
  this.positions=[];
  for(vari=0;i<count;++i){
    this.positions.push(this.calculatePosition(i*this.steps));
  }
  if(this.rigidBody){
    this.initializePosition();
    this.system.addBody(this.rigidBody);
  }
}
```

The `initializePosition` function takes an element from the `positions` array at the `this.counter` value, transforms that value using `initialModelMatrix` from the camera, and moves `rigidBody` to the new calculated position. This is where the cheating happened. This function is also invoked from the `update` function. It simply moves the `rigidBody` parameter to a predefined value; we did not let physics control its motion/trajectory. Initially the counter is `0` to set the initial bullet location.

```
Bullet.prototype.initializePosition=function(){
```

```
  var mat=mat4.create();

mat4.translate(mat,this.initialModelMatrix,this.positions[this.coun
ter]);
  var newPos=jigLib.GLMatrix.multiplyVec3(mat,[0,0,-1]);

this.rigidBody.moveTo(jigLib.Vector3DUtil.create(newPos[0],newPos[1
],newPos[2]));
}
```

The `update` function increments the counter value. The `rigidBody` parameter updates its position by invoking the `initializePosition` function using the incremented counter value. After we are through with iterating over the `positions` array, we do not set the position of `rigidBody` and let the physics engine take over.

We translate `modelMatrix` by the new position of `rigidBody`. Note that for all cases, we read the `rigidBody` coordinate so that when the physics engine takes over, we move the geometry with `rigidBody`.

Earlier, we were toggling the visibility of the bullet when we had iterated over all values of the positions. However, in this case, we wait for the collision to occur, for example, when the bullet hits the ground.

We check the length of the `collisions` array to test the collision. Then, we remove the rigid body from the physics system and also toggle its visibility. The `update` function of the `Bullet` class is as follows:

```
Bullet.prototype.update=function() {
  if(this.rigidBody){
    if(this.counter<this.positions.length-1){
      this.counter=this.counter+1;
      this.initializePosition();

    }
    var pos=this.rigidBody.get_currentState().position;
    mat4.identity(this.modelMatrix);

mat4.translate(this.modelMatrix,this.modelMatrix,vec3.fromValues(po
s[0],pos[1],pos[2]));
    if(this.rigidBody.collisions.length>0){
      this.visible=false;
      this.system.removeBody(this.rigidBody);
      this.rigidBody.collisions=[];
    }
```

```
    }

mat4.scale(this.modelMatrix,this.modelMatrix,vec3.fromValues(5,5,5)
);
  }else{
...
  }
}
```

Let's also walk through the bullet's implementation in the main code. Open the `07-Grenade-Action-Blast-Physics.html` file in your editor.

For the bullet, we initialize a `jigLib.JBox( null, 3, 3, 3 );` object. We set its length, width, and height to `3` and then set its mass to `10`. We do that for all other bullets we initialized. This is done with the help of the following code snippet:

```
function
addStageObject(stageObject,location,rotationX,rotationY,rotationZ){
...
  if(stageObject.name=="bullet"){
    var cube = new jigLib.JBox( null, 3, 3, 3 );
    cube.set_mass(10);
    stageObject.rigidBody = cube;
    stageObject.system=system;
    bullets.push(stageObject);
    for(var j=0;j<24;++j){
      var bullet=stageObject.clone();
      bullet.camera=cam;
      bullets.push(bullet);
      var cube = new jigLib.JBox( null, 3, 3, 3 );
      bullet.rigidBody = cube;
      cube.set_mass(10);
      cube.set_movable(true);
      bullet.system=system;
      stage.addModel(bullet);
    }
  }
...
}
```

Extending our terrain with physics

In our existing code packets, we have a terrain which is a simple plain, with no bumps or plateaus or dips. Hence, we use an object of the `jigLib.Plain` class to represent it in our physics world. This section intends to end what we started with a true physics terrain. The following screenshot shows you the terrain that we will be targeting to initialize in the physics world. It has a bump that starts from the center of the terrain:

We can easily add this bump to the geometry, but it will take some effort to simulate it in our physics world. So, first we add this bump to our geometry. Open `PlaneGeometry.js` from the `primitive` folder in your text editor. In our previous code, we set the *z* axis value to `0` as shown in the following code:

```
this.vertices.push(x);
this.vertices.push(- y);
this.vertices.push(0);
```

The preceding code simply denoted that the geometry had no bumps, and all segments had a height of 0. However, now we simply add some logic to calculate the height for a particular segment. The following constructor takes a function name as a parameter; the function takes the (`x`, `y`) coordinates as parameters and calculates the height for a particular segment. If the function name is not set (that is, it is null or undefined), then the *z* value is set to `0` for all segments:

```
PlaneGeometry = inherit(Geometry, function (width, height,
widthOfSegments, heightOfSegments,calculateHeight)
```

```
{
...
  if(this.calculateHeight){

    this.vertices.push(this.calculateHeight(x,-y));
  }else{

  this.vertices.push(0);
  }
...
}
```

Open `Plane.js` from `primitive/game` in your text editor to see the sample implementation of `calculateHeight`. In the constructor of `PlaneGeometry`, we pass the `modifyGeometry` function of the `Plane` class as shown in the following code snippet:

```
Plane= inherit(StageObject, function (width, height,
widthOfSegments, heightOfSegments,textureName,modifyHeight){
...
  if(modifyHeight)
    this.geometry=new PlaneGeometry(width, height, widthOfSegments,
heightOfSegments,this.modifyGeometry);
  else
    this.geometry=new PlaneGeometry(width, height, widthOfSegments,
heightOfSegments,null);
...
});
```

The `modifyGeometry` function is a straightforward function. It takes the `x` and `y` values based on the range of the `x` and `y` values and returns the *z* value from 25 to 0. Hence, the `PlaneGeometry` class uses this function to calculate the *z* value of the vertices:

```
Plane.prototype.modifyGeometry=function(x,y){
  if((x>=0&&x<100)&&(y>=0&&y<100)){
    return 25;
  }
  else if((x>=100&&x<150)&&(y>=100&&y<150)){
    return 20;
  }
  else if((x>=150&&x<200)&&(y>=150&&y<200)){
    return 15;
  }
  else if((x>=200&&x<250)&&(y>=200&&y<250)){
    return 10;
```

```
    return 10;
  }
  else if((x>=250&&x<300)&&(y>=250&&y<300)){
    return 5;
  }
  else{
    return 0;
  }
}
```

Modifying a geometry is pretty straightforward, but informing the physics engine of it is a bit of challenge. There is an existing class, `jigLib.JTerrain`; its constructor takes a parameter of the type `ITerrain`.

```
var Jterrain=function(tr)
```

The `ITerrain` interface is not implemented in JigLibJS; we will have to implement it. Basically, the implementation of `ITerrain` informs the JigLib collision detection code of the structure of the terrain. Let's first understand the `ITerrain` interface:

```
public interface ITerrain
 {
  //Min of coordinate horizontally;
  function get_minW():Number;
//Min of coordinate vertically;
  function get_minH():Number;
  //Max of coordinate horizontally;
  function get_maxW():Number;
  //Max of coordinate vertically;
  function get_maxH():Number;
  //The horizontal length of each segment;
  function get_dw():Number;
  //The vertical length of each segment;
  function get_dh():Number;
  //Number of segments horizontally.
  function get_sw():int;
  //Number of segments vertically
  function get_sh():int;
  //the heights of all vertices
  function get_heights(i,j):Array;
  function get_maxHeight():Number;
}
```

If you study the properties, the implementation returns the terrain properties such as minimum height, minimum width, number of segments, and length of

segments. Our terrain already has these parameters defined. The most interesting function is `get_heights(i,j)`, which takes the vertical and horizontal index of the segment and returns its height. However, we do not have this parameter defined in our geometry. So, let's define it first. Open PlaneGeometry.js from the `primitive` folder in your text editor. In our constructor, we define a `this.heights=[[]];` variable:

```
PlaneGeometry = inherit(Geometry, function (width, height,
widthOfSegments, heightOfSegments,calculateHeight)
{
...
  this.heights=[[]];
...
  if(this.calculateHeight){
    this.heights[i][j]=this.calculateHeight(x,-y);
    this.vertices.push(this.heights[i][j]);
  }else{
    this.heights[i][j]=0;
    this.vertices.push(0);
  }
...
}
```

After we calculate the height for a vertex and a particular segment, we store its values in the heights array (`this.heights[i][j]=this.calculateHeight(x,-y);`).

We also added a property to return the height of a particular segment (`getHeights(i,j)`):

```
PlaneGeometry.prototype.getHeights=function(i,j){
  returnthis.heights[i][j];
}
```

Now, time to see the implementation of the `ITerrain` interface. Open `TerrainData.js` from `primitive/game` in your editor. It takes nearly the same parameters as our `PlaneGeometry` object took (`width`, `height`, `segmentsW`, `segmentsH`, `geometry`), with one extra parameter: the object of the terrain, `geometry`. The constructor first calculates `minW`, `minH`, `maxW`, and `maxH`. They are the starting and ending coordinates of the terrain and are half the width and height in each direction. The `this._dh` and `this._dw` parameters return the length of each segment. The `get_heights(i,j)` function reads the height for a segment from the `getHeight(i,j)` geometry and returns its height. The

following is the code snippet from the `TerrainData.js` file:

```
TerrainData = function (width,height,segmentsW,segmentsH,geometry)
{
  var textureX= width  2;
  var textureY = height  2;
  //Min of coordinate horizontally;
  this._minW=-textureX;
  //Min of coordinate vertically;
  this._minH= -textureY;
  //Max of coordinate horizontally;
  this._maxW=textureX;
  //Max of coordinate vertically;
  this._maxH=textureY;
  //The horizontal length of each segment;
  this._dw=width  segmentsW;
  /The vertical length of each segment;
  this._dh=height  segmentsH;;
  /the heights of all vertices
  this._heights=[[]];
  this._segmentsW=segmentsW;
  this._segmentsH=segmentsH;
  this.geomtery=geometry;
   this._maxHeight=10;
}
TerrainData.prototype.get_minW=function() {
  return this._minW;
}
TerrainData.prototype.get_minH=function() {
  return this._minH;
}
TerrainData.prototype.get_maxW=function() {
  return this._maxW;
}
TerrainData.prototype.get_maxH=function() {
  return this._maxH;
}
TerrainData.prototype.get_dw=function() {
  return this._dw;
}
TerrainData.prototype.get_dh=function() {
  return this._dh;
}
TerrainData.prototype.get_sw=function() {
  return this._segmentsW;
}
TerrainData.prototype.get_sh=function() {
```

```
  return this._segmentsH;
}
TerrainData.prototype.get_heights=function(i1,j1) {
  return this.geomtery.getHeights(i1,j1);
}
TerrainData.prototype.get_maxHeight=function(){
  return this._maxHeight;
}
```

The preceding class holds the complete information of the terrain, height, width, segment information, and also the depth of each segment. We pass the `TerrainData` object to the physics engine and the engine checks for collisions with the provided information.

Open `Plane.js` from `primitive/game` to see the use of the `TerrainData` object. We have added a new function, `initializeRigidBody`. The following is a code snippet from the `Plane.js` file:

```
Plane= inherit(StageObject, function (width, height,
widthOfSegments, heightOfSegments,textureName,modifyHeight){
...
  this.geometry=new PlaneGeometry(width, height, widthOfSegments,
heightOfSegments,this.modifyGeometry);
...
});
The preceding code initializes the geometry, and it PlaneGeometry
constructor initializes the heights[[]] array.
Plane.prototype.initializeRigidBody=function(){
  var pos=jigLib.Vector3DUtil.create( 0, 0,0,0 );
  var matrix3D = new jigLib.Matrix3D();
  matrix3D.appendRotation(0, jigLib.Vector3DUtil.X_AXIS);
  var terrain=new
TerrainData(this.width,this.height,this.ws,this.wh,this.geometry);
  this.rigidBody=new jigLib.JTerrain(terrain);
  this.rigidBody.moveTo(pos);
  this.rigidBody.setOrientation(matrix3D);
}
```

The preceding code initializes the `TerrainData` object with the same properties that we initialized the geometry with (`new TerrainData(this.width, this.height,this.ws,this.wh, this.geometry);`). Then, we create a new `jigLib.JTerrain(terrain);` object and assign it to the `rigidBody` object.

Now, when you run `07-Terrain-Physics-Modified.html` in your browser and

you fire bullets or drop grenades on the bump, you will find that it is an active physics static object.

Implementing height maps

Our implementation to modify geometry is very simple (`Plane.prototype.modifyGeometry`), we just check the value of `x` and `y` and return a hard coded value of *z*, but this not how most implementations work. Our geometry can be modified using the following code snippet:

```
Plane.prototype.modifyGeometry=function(x,y){
...
  else if((x>=100&&x<150)&&(y>=100&&y<150)){
    return 20;
  }
...
}
```

The correct implementation is using height maps. A height map is a simple 2D grayscale image as shown in the following image. The image has shades of gray, one base color with light and dark areas. The dark areas denote heights and lighter areas denote dips:

The height map is not the same size as the terrain; however, we can always map a coordinate of the terrain to the height map:

x coordinate on the height map = width HM/width Terrain * x value terrain

y coordinate on the height map = height HM/height Terrain * y value terrain

After we get the corresponding coordinate on the height map, we can calculate the height at a location by retrieving the color value on the coordinate and multiplying it with the maximum height difference of the terrain:

Height at a coordinate = min_height + (color value / (max_color - min_color)) * (max_height-min_height)

So, you can modify the `modifyGeometry` code to read the height map and return the *z* value for a particular segment.

Summary

Realistic games are impossible to create without physics engines. This chapter introduced you to the most exciting areas in game development using WebGL and physics. We started our journey by creating terrain geometry and finished it by making the geometry an active physics static rigid body. We also covered how a physics engine is decoupled from the rendering code and demonstrated how 3D graphics objects are linked to the physics object. The concepts such as collisions, forces, and impulse are in the core of each game.

We used JigLib as a reference library to implement physics in our game.

In the next chapter, we will introduce you to your enemy Mr. Green and animate his body using bones and joints.

Chapter 8. Skinning and Animations

Our world in 5000 AD is incomplete without our mutated human being Mr. Green. Our Mr. Green is a rigged model, exported from Blender. All famous 3D games from *Counter Strike* to *World of Warcraft* use skinned models to give the most impressive real world model animations and kinematics. Hence, our learning has to now evolve to load Mr. Green and add the same quality of animation in our game.

We will start our study of character animation by discussing the skeleton, which is the base of the character animation, upon which a body and its motion is built. Then, we will learn about skinning, how the bones of the skeleton are attached to the vertices, and then understand its animations. In this chapter, we will cover:

- Basics of a character's skeleton
- Basics of skinning
- Loading a rigged JSON model
- Animating a rigged JSON model
- Exporting models from 3D software in JSON

Understanding the basics of a character's skeleton

A character's skeleton is a posable framework of bones. These bones are connected by articulated joints, arranged in a hierarchical data structure. The skeleton is generally rendered and is used as an invisible armature to position and orient a character's skin.

The joints are used for relative movement within the skeleton. They are represented by a 4 x 4 linear transformation matrices (combination of rotation, translation, and scale). The character skeleton is set up using only simple rotational joints as they are sufficient to model the joints of real animals.

Every joint has limited **degrees of freedom** (**DOFs**). DOFs are the possible ranges of motion of an object. For instance, an elbow joint has one rotational DOF and a shoulder joint has three DOFs, as the shoulder can rotate along three perpendicular axes. Individual joints usually have one to six DOFs. Refer to the link http://en.wikipedia.org/wiki/Six_degrees_of_freedom to understand different degrees of freedom.

A joint local matrix is constructed for each joint. This matrix defines the position and orientation of each joint and is relative to the joint above it in the hierarchy. The local matrices are used to compute the world space matrices of the joint, using the process of forward kinematics. The world space matrix is used to render the attached geometry and is also used for collision detection.

The digital character skeleton is analogous to the real-world skeleton of vertebrates. However, the bones of our digital human character do have to correspond to the actual bones. It will depend on the level of detail of the character you require. For example, you may or may not require cheek bones to animate facial expressions.

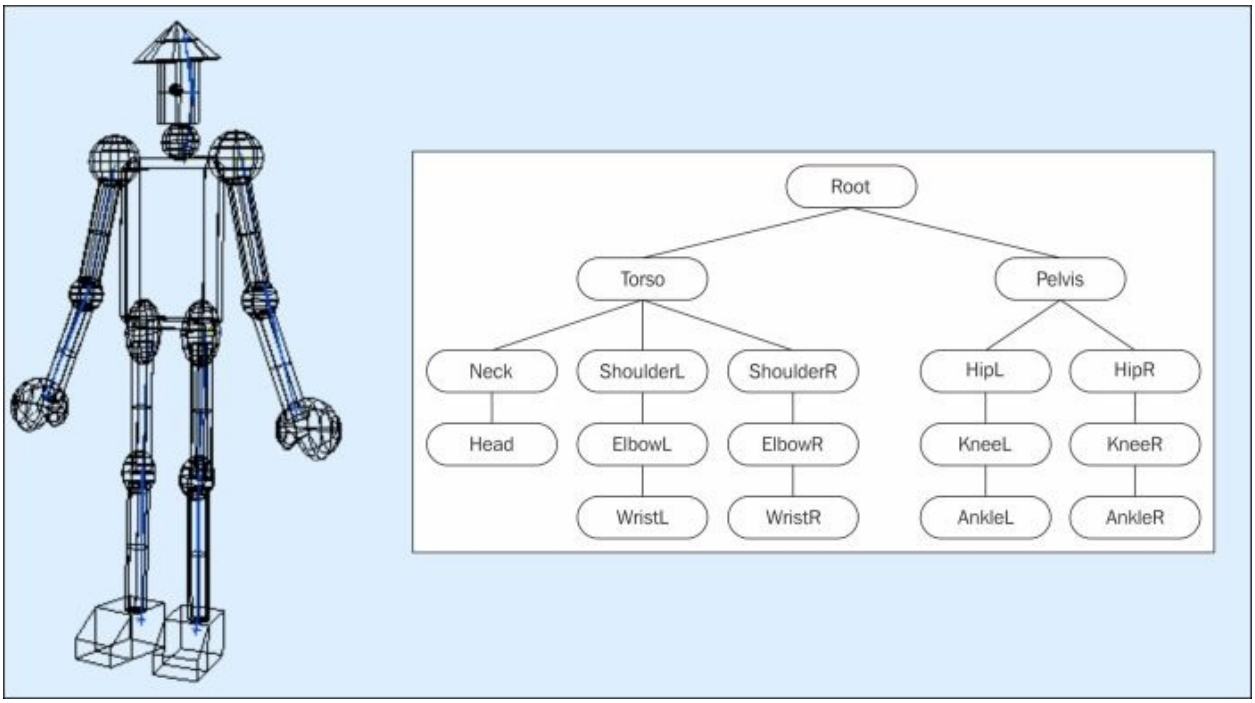
Skeletons are not just used to animate vertebrates but also mechanical parts such as doors or wheels.

Comprehending the joint hierarchy

The topology of a skeleton is a tree or an open-directed graph. The joints are connected up in a hierarchical fashion to the selected root joint. The root joint has no parent of itself and is presented in the model JSON file with the parent value of `-1`. All skeletons are kept as open trees without any closed loops. This restriction though does not prevent kinematic loops.

Each node of the tree represents a joint, also called bones. We use both terms interchangeably. For example, the shoulder is a joint, and the upper arm is a bone, but the transformation matrix of both objects is same. So mathematically, we would represent it as a single component with three DOFs. The amount of rotation of the shoulder joint will be reflected by the upper arm's bone.

The following figure shows simple robotic bone hierarchy:



Understanding forward kinematics

Kinematics is a mathematical description of a motion without the underlying physical forces. Kinematics describes the position, velocity, and acceleration of an object. We use kinematics to calculate the position of an individual bone of the skeleton structure (skeleton pose). Hence, we will limit our study to position and orientation. The skeleton is purely a kinematic structure. Forward kinematics is used to compute the world space matrix of each bone from its DOF value. Inverse kinematics is used to calculate the DOF values from the position of the bone in the world.

Let's dive a little deeper into forward kinematics and study a simple case of bone hierarchy that starts from the shoulder, moves to the elbow, finally to the wrist. Each bone/joint has a local transformation matrix, `this.modelMatrix`. This local matrix is calculated from the bone's position and rotation. Let's say the model matrices of the wrist, elbow, and shoulder are $\texttt{this.modelMatrix}_{\text{wrist}}$, $\texttt{this.modelMatrix}_{\text{elbow}}$, and $\texttt{this.modelMatrix}_{\text{shoulder}}$ respectively. The world matrix is the transformation matrix that will be used by shaders as the model matrix, as it denotes the position and rotation in world space.

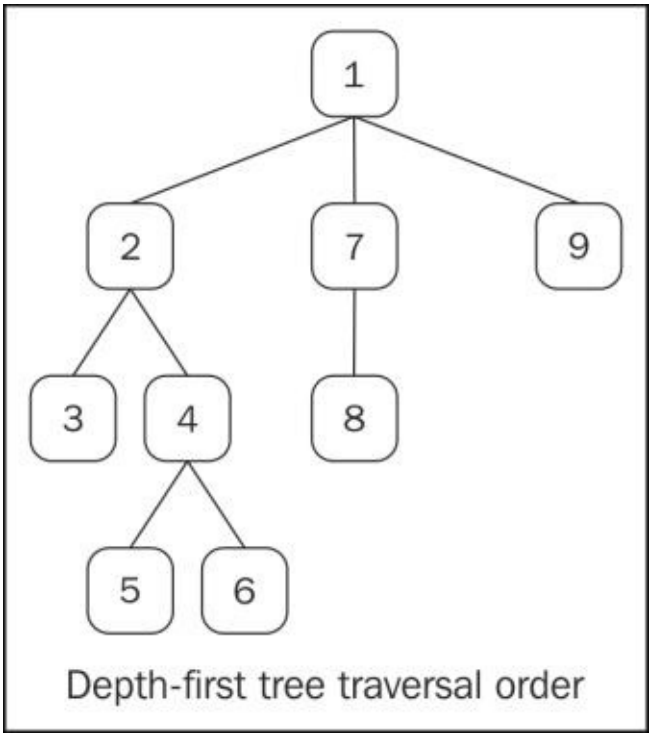
The world matrix for a wrist will be:

$\text{this.worldMatrix}_{\text{wrist}} = \text{this.worldMatrix}_{\text{elbow}} * \text{this.modelMatrix}_{\text{wrist}}$

The world matrix for an elbow will be:

$\text{this.worldMatrix}_{\text{elbow}} = \text{this.worldMatrix}_{\text{shoulder}} * \text{this.modelMatrix}_{\text{elbow}}$

If you look at the preceding equations, you will realize that to calculate the exact location of a wrist in the world space, we need to calculate the position of the elbow in the world space first. To calculate the position of the elbow, we first need to calculate the position of shoulder. We need to calculate the world space coordinate of the parent first in order to calculate that of its children. Hence, we use depth-first tree traversal to traverse the complete skeleton tree starting from its root node.



A depth-first traversal begins by calculating `modelMatrix` of the root node and traverses down through each of its children. A child node is visited and subsequently all of its children are traversed. After all the children are visited, the control is transferred to the parent of `modelMatrix`. We calculate the world matrix by concatenating the joint parent's world matrix and its local matrix. The computation of calculating a local matrix from DOF and then its world matrix from the parent's world matrix is defined as forward kinematics.

Let's now define some important terms that we will often use:

- **Joint DOFs**: A movable joint movement can generally be described by six DOFs (three for position and rotation each). DOF is a general term:

  ```
  this.position = vec3.fromValues(x, y, z);
  this.quaternion = quat.fromValues(x, y, z, w);
  this.scale = vec3.fromValues(1, 1, 1);
  ```

  We use quaternion rotations to store rotational transformations to avoid issues such as gimbal lock, as explained in Chapter 5, *Camera and User Interaction*. The quaternion holds the DOF values for rotation around the *x*, *y*, and *z* values.

- **Joint offset**: Joints have a fixed offset position in the parent node's space. When we skin a joint, we change the position of each joint to match the mesh. This new fixed position acts as a pivot point for the joint movement. The pivot point of an elbow is at a fixed location relative to the shoulder joint. This position is denoted by a vector position in the joint local matrix and is stored in *m31*, *m32*, and *m33* indices of the matrix. The offset matrix also holds initial rotational values.

Understanding the basics of skinning

The process of attaching a renderable skin to its articulated skeleton is called skinning. There are many skinning algorithms depending on the complexity of the task. However, for gaming, the most common algorithm is smooth skinning. Smooth skinning is also known as multi-matrix skinning, blended skinning, or linear blend skinning.

Simple skinning

Binding is a term common in skinning. It refers to the initial assignment of vertices of a mesh to underlying joints and then assigning the relevant information to the vertices. By using simple skinning, we attach every vertex in our mesh to exactly one joint. When we change the orientation of any joint in the skeleton, or in other words, when the skeleton is posed, the vertices are transformed using the joint's world matrix. Hence, if the vertex is attached to a single joint, then it is transformed using the equation $v'=v.mjoint$ of the world space matrix.

Simple skinning is not adequate for complex models. It defines that a vertex is attached to exactly one joint. For example, a vertex at the elbow of your articulated arm is affected by two bones, the lower arm and the upper arm. The transformation of that vertex should be affected by the joint matrices of both bones.

Smooth skinning

Smooth skinning is an extension of simple skinning. We can attach a vertex with more than one joint. Each attachment with a joint will be provided by a weight value. The key point is that the sum total of all weights affecting a vertex is 1 as shown in the following formula:

$\sum w_i=1$, $w_1+w_2+w_3+w_4.....w_n=1$

The final vertex's transformed position is the weighted average of the initial vertex position transformed by each of the attached joints. However, before deriving the formula for the vertex position, let's first understand the concept of the binding matrix. The B_i matrix for the *i* joint is a matrix of the transformation of the coordinate joint local space to skin local space. To transform a point from skin local space to joint local space, we use B^{-1}_i, the inverse of the binding matrix.

The binding matrix

Although, the binding matrix is a simple concept, sometimes it baffles the most intelligent of the minds. When we draw a mesh, each vertex is provided with a position relative to the model's center. It is like the model is centered at the origin of our world. During modeling, we create the skeleton along with the skin, each bone/joint at this point has a zero DOF value, we call this pose a zero pose. However, during the skinning process, we change the position of each joint to match the mesh. This pose is called the binding pose. Note that we change the DOF (position and angles) of each joint to match the vertices. The initial DOF values of the binding pose for each joint form the binding matrix or we can say, the initial joint matrix. This matrix is used to transform any position from joint local space to skin local space. Remember that each vertex is defined in skin local space. Hence, to transform a coordinate from skin local space to joint local space, we use inverse joint matrix B^{-1}_i.

During animation, we change DOF values (position and rotations) of a joint, but these values are in joint local space. Hence, the final vertex is transformed using $M_i=B^{-1}_iW_i$ where W_i is the joint matrix in the world space. Hence first, we transform a vertex from skin local space to joint local space and then, we transform it using the joint's world space matrix. For a pose or animation frame,

we calculate the M_i for all joints and then pass this final transformation matrix as a uniform to the shader so that we do not have to recalculate it for other vertices in the same joint, as shown in the following code snippet:

```
// compute the offset between the current and the original
transform.
mat4.mul(offsetMatrix,this.bones[ b ].skinMatrix,
this.boneInverses[ b ]);
```

The final vertex transformation

The final vertex transformation is the weighted average of the initial vertex position transformed by each of the attached joints, $v'=\sum w_i*v*M_i$, where $Mi=B^{-1}{}_iWi$ and *wi* are the weight value of a joint for vertex.

In most cases, a vertex is shared between two bones and maximum of four bones. Hence for simplicity, our code only handles skinned models whose vertices are shared with a maximum of two joints.

```
vec4 skinVertex = vec4(aVertexPosition, 1.0);
vec4 skinned = boneMatX  skinVertex  skinWeight.x;
    skinned   += boneMatY  skinVertex  skinWeight.y;
```

In the preceding code, `boneMatX` is the offset matrix for bone X with its contributing weight in `skinWeight.x`, and `boneMatY` is an offset matrix of the second bone with its contributing weight in `skinWeight.y`.

The transformation computation is performed in the vertex shader.

The final normal transformation

We would also need to transform our vertex normals as lighting calculation uses vertex normals. The normals are treated in a similar fashion to vertices, but as normals only specify direction and not position and are of unit length, we first calculate the weighted average and then multiply the normal with `skinMatrix` to avoid one extra multiplication step, as shown in the following code snippet:

```
mat4 skinMatrix = skinWeight.x  boneMatX;
skinMatrix += skinWeight.y  boneMatY;
vec4 skinnedNormal = skinMatrix  vec4( aVertexNormal, 0.0);
transformedNormal = vec3(nMatrix  skinnedNormal);
```

Loading a rigged JSON model

We will first understand how the bone DOFs and skinning information is encoded in the three.js JSON file format (Version 3.1). Then we will modify our code to load the data. The JSON file is exported from Blender. We will learn how to export the JSON file in the *Exporting models from 3D software in JSON* section.

Understanding JSON file encoding

The JSON file contains bone DOF values and their corresponding skinning information. Open `model/obj/mrgreen.json` in your favorite text editor. The file has now four new arrays: `bones`, `skinIndices`, `skinWeights`, and `animation`. We will discuss the `animation` array in the *Animating a rigged JSON model* section.

The `bones` array contains the DOF information. It holds the binding matrix and its parent's information, as shown in the following code:

```
"bones" : [{"parent":-1, "name":"Back", "pos":[0.000000, -0.123622,
-0.149781], "rotq":[0,0,0,1]}, {}, {}.......];
```

Each element of the `bones` array holds the following four elements:

- `parent`: This element holds the hierarchical information of the skeleton. Each bone holds its parent's index. The root bone has a parent index of -1, denoting it does not have any parent.
- `name`: This element holds the name of the bone.
- `pos`: This element is a vector and holds the position of each bone with respect to its parent.
- `rotq`: Each bone's rotation is expressed as a quaternion rotation (*x*, *y*, *z*, and *w*) with respect to its parent. Chapter 5, *Camera and User Interaction*, has a description of quaternion rotations.

For each vertex(*x*, *y*, *z*, *x1*, *y1*, and *z1*) in the `vertices` array, there are two values defined in the `skinIndices` (*a*, *b*, *a1*, and *b1*) and `skinWeights` (*a*, *b*, *a1*, and *b1*) arrays. We had discussed earlier in the *Understanding the basics of skinning* section that we will use a smooth skinning algorithm to store weights and skinning information. The `three.js` JSON model (https://github.com/mrdoob/three.js/wiki/JSON-Model-format-3.1) allows only two attached bones per vertex. Hence for each vertex, we will have two corresponding `skinIndices` and `skinWeights` defined. Although a vertex may be associated with more than two bones, it is not advisable or even not required in gaming. It would rarely happen that a vertex is affected by three bones simultaneously. The `skinIndices` array holds the index of the bone in the `bones` array.

```
vertices:[x,y,z,x1,y1,z1,x2,y2,z2.............xn,yn,zn];
skinIndices:[a,b,a1,b1,a2,b2.........an,bn];
```

```
skinWeights:[z,w,z1,w1,z2,w2.......zn,wn];
bones:[]
```

The preceding arrays denote the following:

- The vertices `x`, `y`, and `z` are attached to the bones[a] and bones[b] with weights z and w.
- The vertices `x1`, `y1`, and `z1` are attached to the bones[a1] and bones[b1] with weights `z1` and `w1`.
- The vertices `x2`, `y2`, and `z2` are attached to the bones[a2] and bones[b2] with weights `z2` and `w2`.

A vertex might be associated with a single bone, but we will still have two skin indices (a and b) and two skin weights (z and w) associated with it. In this case, one of the skin weights (z and w) will be `1` and the other would be `0`, denoting that only one of the bones will affect the vertex.

Loading the rigged model

We will first modify our parsing algorithm to accommodate our newly discovered arrays.

Open `primitive/parseJSON.js` in your favorite text editor. We have added a new `parseSkin` function as follows:

```
function parseSkin(data, geometry) {
  var i, l, x, y, z, w, a, b, c, d;
  if ( data.skinWeights ) {
    for ( i = 0, l = data.skinWeights.length; i < l; i += 2 ) {
      x = data.skinWeights[ i     ];
      y = data.skinWeights[ i + 1 ];
      z = 0;
      w = 0;
      geometry.skinWeights.push(x);
      geometry.skinWeights.push(y);
      geometry.skinWeights.push(z);
      geometry.skinWeights.push(w);
    }
  }
  if ( data.skinIndices ) {
    for ( i = 0, l = data.skinIndices.length; i < l; i += 2 ) {
      a = data.skinIndices[ i     ];
      b = data.skinIndices[ i + 1 ];
      c = 0;
      d = 0;
      geometry.skinIndices.push(a);
      geometry.skinIndices.push(b);
      geometry.skinIndices.push(c);
      geometry.skinIndices.push(d);
    }
  }

  geometry.bones = data.bones;
  geometry.animation = data.animation;
}
```

The function simply iterates over the `skinIndices` and `skinWeights` arrays in our data object and stores the four values for each vertex in the corresponding geometry arrays. Note that although our JSON array has two bones per vertex, we still store four values (the last two values as zero, `{c = 0; d = 0;}`), so that our geometry class can handle data with two to four bones per vertex.

We also save the data for bones and animation information in the geometry object.

Enhancing the StageObject class

Our `StageObject` class had two shortcomings:

- It did not have any provision to handle child objects or tree hierarchy.
- We used the rotation matrix but we know that our bone object in the `bones` array uses quaternion rotations.

The following code shows the earlier use of `modelMatrix` to store rotations in the *x*, *y*, and *z* axes:

```
StageObject.prototype.update=function(steps) {
  mat4.identity(this.modelMatrix);

  mat4.translate(this.modelMatrix, this.modelMatrix,
this.location);

  mat4.rotateX(this.modelMatrix, this.modelMatrix, this.rotationX);
  mat4.rotateY(this.modelMatrix, this.modelMatrix, this.rotationY);
  mat4.rotateZ(this.modelMatrix, this.modelMatrix, this.rotationZ);
}
```

Let's walk through the changes we have made to overcome the shortcomings. Open `primitive/StageObject.js` in your editor, and take a look at the following code:

```
StageObject=function() {
...
  this.parent = undefined;
  this.children = [];
  this.up = vec3.fromValues( 0, 1, 0 );
  this.position = vec3.create();
  this.quaternion = quat.create();
  this.scale = vec3.fromValues(1,1,1 );
  this.matrixWorld = mat4.create();
  this.matrixAutoUpdate = true;
  this.matrixWorldNeedsUpdate = true;
  this.visible = true;
};
```

First, we added a few variables such as `quaternion` to hold the rotation DOF, `location` has been renamed to `position`, and new variables, `scale` and

matrixWorld, have been added. If stageObject is the child object, then the final matrix, worldMatrix, is the concatenation of its parent, matrixWorld, and modelMatrix.

The parent object and the children array have been added to hold the parent and children information.

Two new variables, matrixAutoUpdate and matrixWorldNeedsUpdate, have been added to reduce the possible computation time. Basically in our previous code packets, we were calculating modelMatrix of each StageObject on every animation frame. However, now, we will only calculate the matrices if any of the DOFs (scale, quaternion, and position) change. On any DOF update, we will set the matrixAutoUpdate and matrixWorldNeedsUpdate values to false, then only modelMatrix and matrixWorld will be recalculated.

```
StageObject.prototype.rotate=function(radianX,radianY,radianZ) {
  quat.rotateX(this.quaternion,this.quaternion,radianX);
    quat.rotateY(this.quaternion,this.quaternion,radianY);
    quat.rotateZ(this.quaternion,this.quaternion,radianZ);
  }
  StageObject.prototype.setRotationFromAxisAngle=function ( axis,
angle) {
    // assumes axis is normalized
    quat.setAxisAngle(this.quaternion, axis, angle );
  }
  StageObject.prototype.setRotationFromMatrix= function ( m ) {
    // assumes the upper 3 x 3 of m is a pure rotation matrix (that
is, unscaled)
    quat.fromMat3(this.quaternion, m );
  }
  StageObject.prototype.setRotationFromQuaternion=function ( q ) {
    // assumes q is normalized
    this.quaternion=quat.clone( q );
  }
  StageObject.prototype.rotateOnAxis= function(axis, angle) {
    // rotate object on axis in object space
    // axis is assumed to be normalized
    quat.setAxisAngle(this.quaternion, axis, angle );
  }
  StageObject.prototype.rotateX= function (angle) {
    var v1 = vec3.fromValues( 1, 0, 0 );
        return this.rotateOnAxis( v1, angle );
  }
  StageObject.prototype.rotateY= function (angle) {
    var v1 = vec3.fromValues( 0, 1, 0 );
```

```
      return this.rotateOnAxis( v1, angle );
  }
  StageObject.prototype.rotateZ=function (angle) {
    var v1 = vec3.fromValues( 0, 0, 1 );
      return this.rotateOnAxis( v1, angle );
  }
```

The preceding set of functions either initializes the quaternion or simply updates it with new values. The implementation of the preceding functions uses the `quat` class of the `glMatrix` library.

```
StageObject.prototype.translateOnAxis= function (axis, distance) {
  // translate object by distance along axis in object space
  // axis is assumed to be normalized
  var v1 = vec3.create();
  vec3.copy(v1, axis );
  vec3.transformQuat(v1, v1, this.quaternion);
  vec3.scale(v1, v1, distance);
  vec3.add(this.position, this.position, v1);
  return this;
}
StageObject.prototype.translateX= function () {
  var v1 = vec3.fromValues( 1, 0, 0 );
  return function ( distance ) {
    return this.translateOnAxis( v1, distance );
  };
}();
StageObject.prototype.translateY= function () {
  var v1 = vec3.fromValues( 0, 1, 0 );
  return function ( distance ) {
    return this.translateOnAxis( v1, distance );
  };
}();
StageObject.prototype.translateZ= function () {
  var v1 = vec3.fromValues( 0, 0, 1 );
  return function ( distance ) {
    return this.translateOnAxis( v1, distance );
  };
}();
```

The preceding set of functions translates `StageObject` along the given axis. The key function is `translateOnAxis`, and all other functions are dependent on it.

```
StageObject.prototype.localToWorld= function ( vector ) {
  var v1=vec3.create();
  vec3.transformQuat(v1,vector,this.matrixWorld );
  return v1;
```

```
  return v1;
};
StageObject.prototype.worldToLocal= function () {
  var m1 = mat4.create();
  return function ( vector ) {
    mat4.invert(m1,this.matrixWorld);
    var v1=vec3.create();
    vec3.transformQuat(v1,vector,m1 );
    return v1;
  };
}();
```

The preceding functions transform any vector from the world space to the object's local space and vice versa.

```
StageObject.prototype.add=function ( object ) {
  if ( object === this ) {
    return;
  }

  if ( object.parent !== undefined ) {
    object.parent.remove( object );
  }
  object.parent = this;
  //object.dispatchEvent( { type: 'added' } );
  this.children.push( object );
  // add to scene
};
StageObject.prototype.remove= function ( object ) {
  var index = this.children.indexOf( object );
  if ( index !== - 1 ) {
    object.parent = undefined;
    //object.dispatchEvent( { type: 'removed' } );
    this.children.splice( index, 1 );
  }
}
```

The `add` function pushes the object on its `children` array and sets its `parent` value to itself after verifying that the child object maintains an open-graph structure. It first checks if the object has a parent and then it removes the object from its parent's list by invoking the `remove` function of its parent.

The `remove` function unsets the `parent` of the object and deletes it from its `children` array.

```
StageObject.prototype.traverse= function ( callback ) {
```

```
  callback( this );
  for ( var i = 0, l = this.children.length; i < l; i ++ ) {
    this.children[ i ].traverse( callback );
  }
}
StageObject.prototype.getObjectById=function ( id, recursive ) {
  for ( var i = 0, l = this.children.length; i < l; i ++ ) {
    var child = this.children[ i ];
    if ( child.id === id ) {
      return child;
    }
    if ( recursive === true ) {
      child = child.getObjectById( id, recursive );
      if ( child !== undefined ) {
        return child;
      }
    }
  }
  return undefined;
}
StageObject.prototype.getObjectByName= function ( name, recursive )
{
  for ( var i = 0, l = this.children.length; i < l; i ++ ) {
    var child = this.children[ i ];
    if ( child.name === name ) {
      return child;
    }
    if ( recursive === true ) {
      child = child.getObjectByName( name, recursive );
      if ( child !== undefined ) {
        return child;
      }
    }
  }
  return undefined;
}
StageObject.prototype.getChildByName= function ( name, recursive )
{
  return this.getObjectByName( name, recursive );
}
StageObject.prototype.getDescendants=function ( array ) {
  if ( array === undefined ) array = [];
  Array.prototype.push.apply( array, this.children );
  for ( var i = 0, l = this.children.length; i < l; i ++ ) {
    this.children[ i ].getDescendants( array );
  }
  return array;
}
```

```
}
```

We have also added traversal functions to locate the child objects either by ID or by name. The key function is `traverse`; it calls itself recursively followed by the depth-first search algorithm.

```
StageObject.prototype.updateMatrix=function () {
  mat4.identity(this.modelMatrix);
  mat4.fromQuat(this.modelMatrix,this.quaternion);
  mat4.scale(this.modelMatrix,this.modelMatrix,this.scale);
  this.modelMatrix[12]=this.position[0];
  this.modelMatrix[13]=this.position[1];
  this.modelMatrix[14]=this.position[2];
  this.matrixWorldNeedsUpdate = true;
}
```

The preceding function is the most significant change we have done from the previous code. Earlier, we were using rotational matrices to compute the object's transformation matrix, but now we are using the quaternion to calculate the model matrix (`mat4.fromQuat(this.modelMatrix, this.quaternion)`). Then, we apply shear transformation and scale our object with the provided scale vector. Then we simply place the `position` vector in *m31*, *m32*, and *m33* of our transformation matrix.

```
StageObject.prototype.updateMatrixWorld=function ( force ) {
  if ( this.matrixAutoUpdate === true ) this.updateMatrix();
  if ( this.matrixWorldNeedsUpdate === true || force === true ) {
    if ( this.parent === undefined ) {
      this.matrixWorld.copy( this.modelMatrix );
    } else {
    mat4.mul(this.matrixWorld, this.parent.matrixWorld,
this.modelMatrix);
    }
    this.matrixWorldNeedsUpdate = false;
    force = true;
  }
  // update children
  for ( var i = 0, l = this.children.length; i < l; i++ ) {
    this.children[ i ].updateMatrixWorld( force );
  }
}
StageObject.prototype.update=function(steps) {
  this.updateMatrixWorld();
}
```

Another interesting function is the `updateMatrixWorld` function. It first invokes `updateMatrix`; if `matrixAutoUpdate` is `true`, the function then checks for the value of `parent`. If `parent` is not defined, then `modelMatrix` is copied to `matrixWorld`; otherwise, `matrixWorld` for that object is computed by concatenating the parent's `matrixWorld` matrix and the object's `modelMatrix` (`mat4.mul(this.matrixWorld,this.parent.matrixWorld,this.modelMatrix)`) Then, we iterate over all the children of the object to compute their new world matrix. We have also updated our `update` function. It invokes `updateMatrixWorld` when it is invoked from our main control code.

Implementing the bone class

Though the bones are never rendered, we will treat them as stage objects, as all the transformations applied to stage objects are applied to the bones too. We do not add them to our stage but we surely inherit the `StageObject` class to achieve the desired functionality. Open `primitive/Bone.js` in your editor and examine the following code snippet:

```
Bone= inherit(StageObject, function (belongsToSkin ) {
  superc(this);
  var d = new Date();
  this.id ="id-"+d.getTime();
  this.skin = belongsToSkin;
  this.skinMatrix = mat4.create();
});
```

The `Bone` class inherits `StageObject` and has all its properties. We have added two more variables to the class, `skin` and `skinMatrix`. The `skin` variable holds the reference to the `Geometry` class, where the bone belongs to. The `skinMatrix` variable is very similar to `matrixWorld`, but it holds the world space transformation of the bone. These variables are updated using the following function:

```
Bone.prototype.update=function ( parentSkinMatrix, forceUpdate ) {
  // update local
  if ( this.matrixAutoUpdate ) {
    forceUpdate |= this.updateMatrix();
  }
  // update skin matrix
  if ( forceUpdate || this.matrixWorldNeedsUpdate ) {
    if( parentSkinMatrix ) {
      mat4.mul(this.skinMatrix,parentSkinMatrix, this.modelMatrix);
      //console.log(parentSkinMatrix);
    } else {
```

```
    } else {
      mat4.copy(this.skinMatrix,this.modelMatrix );
    }
    this.matrixWorldNeedsUpdate = false;
    forceUpdate = true;
  }
  // update children
  var child, i, l = this.children.length;
  for ( i = 0; i < l; i ++ ) {
    this.children[ i ].update( this.skinMatrix, forceUpdate );
  }
}
```

We have added a new `update` function to the `Bone` class. It takes the parent object's skin matrix and concatenates it to `modelMatrix` to compute a bone's `skinMatrix` (`mat4.mul(this.skinMatrix,parentSkinMatrix, this.modelMatrix)`). Then, it invokes the `update` functions of all its child bones with its `skinMatrix` as a parameter.

Note

The `update` function is very similar to the `updateMatrixWorld` function of the `StageObject` class, except that it uses a different variable, `skinMatrix`, and not `matrixWorld`. We did this to differentiate a bone from other renderable objects(`StageObject`) and their properties.

Implementing the RiggedMesh class

The `RiggedMesh` class is where the complete magic happens. It is a renderable object that inherits `StageObject` but has its `children` array populated with `bones` initialized from the JSON file.

Open `primitive/RiggedMesh.js` in your favorite editor.

```
RiggedMesh= inherit(StageObject, function (geometry) {
  superc(this);
  this.identityMatrix = mat4.create();
  this.bones = [];
  this.boneMatrices = [];
  this.skinIndexBuffer=null;
  this.skinWeightBuffer=null;
});
```

The `RiggedMesh` class has four new variables defined as follows:

- `this.bones[]`: This array holds all the `bone` objects for that mesh.
- `this.boneMatrices[]`: This array holds the transformation matrix of all bones flattened into a common array. For example, elements 0 to 15 will hold the transformation matrix of bone 0, elements 16 to 31 will hold the transformation matrix of bone 1, and so on.
- `this.skinIndexBuffer`: This variable holds the reference to the Vertex Buffer Object, which stores the `skinIndices` data.
- `this.skinWeightBuffer`: This variable holds the reference to the vertex buffer object, which stores the `skinWeights` data.

The `loadObject` function of the `RiggedMesh` class is defined as follows:

```
RiggedMesh.prototype.loadObject= function (data) {
...
  this.geometry=parseJSON(data);
  parseSkin(data,this.geometry);
...
...
  var b, bone, gbone, p, q, s;
  if ( this.geometry && this.geometry.bones !== undefined ) {
    for ( b = 0; b < this.geometry.bones.length; b ++ ) {
      gbone = this.geometry.bones[ b ];
      p = gbone.pos;
      q = gbone.rotq;
      s = gbone.scl;
      bone = this.addBone();
      bone.name = gbone.name;
      bone.position=vec3.fromValues( p[0], p[1], p[2] );
      bone.quaternion=quat.fromValues( q[0], q[1], q[2], q[3] );
      if ( s !== undefined ) {
        bone.scale=vec3.fromValues( s[0], s[1], s[2] );
      } else {
        bone.scale=vec3.fromValues( 1, 1, 1 );
      }
    }
    for ( b = 0; b < this.bones.length; b ++ ) {
      gbone = this.geometry.bones[ b ];
      bone = this.bones[ b ];
      if ( gbone.parent === -1 ) {
        this.add( bone );
      } else {
        this.bones[ gbone.parent ].add( bone );
      }
    }
    //
    var nBones = this.bones.length;
```

```
    var nBones = this.bones.length;
    this.boneMatrices = new Float32Array( 16 * nBones );
    this.pose();
  }
}
```

The `loadObject` function overrides the `StageObject` class' `loadObject` function. It does everything that `StageObject` does, such as it invokes `parseJSON` and it initializes materials. It also parses the skin and populates the geometry with the `skinIndices` and `skinWeights` data.

Then, it iterates over the `bones` array and initializes a bone object for each element of the array. It reads the `position`, `quaternion` and `scale` values from the `bones` array element and adds the newly created bone object to the `bones` array.

Then, it creates the tree hierarchy. It iterates over all bone elements and if the value of `parent` of the bone is `-1`, it adds the bone as its child element; otherwise, it adds the bone to the corresponding bone's parent, `(this.bones[ gbone.parent ].add( bone );)`. The `parent` attribute of the bone element holds the indices of the bone. Hence, we first retrieve the parent bone object by using the `this.bones[ gbone.parent]` code and then add the bone to its parent.

It invokes the `this.pose()` function after initializing the bone objects, which in turn invokes `updateMatrixWorld`, which creates the initial world space matrix for each bone. The `createBuffers` function is defined as follows:

```
RiggedMesh.prototype.createBuffers=function(gl) {
...
  this.skinIndexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.skinIndexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(this.geometry.skinIndices), gl.STATIC_DRAW);
  this.skinIndexBuffer.itemSize = 4;
  this.skinIndexBuffer.numItems =
this.geometry.skinIndices.length/4;
  this.skinWeightBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.skinWeightBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(this.geometry.skinWeights), gl.STATIC_DRAW);
  this.skinWeightBuffer.itemSize = 4;
  this.skinWeightBuffer.numItems =
this.geometry.skinWeights.length/4;
...
```

```
}
```

The preceding function initializes the vertex buffer objects for `skinIndices` and `skinWeights`. It then loads the data in the GPU memory using the ESSL function, `gl.bufferData`. The `addBone` function is defined as follows:

```
RiggedMesh.prototype.addBone = function( bone ) {
  if ( bone === undefined ) {
    bone = new Bone( this );
  }
  this.bones.push( bone );
  return bone;
};
RiggedMesh.prototype.updateMatrixWorld = function () {
  var offsetMatrix = mat4.create();
  return function ( force ) {
    this.matrixAutoUpdate && this.updateMatrix();
    // update matrixWorld
      if ( this.matrixWorldNeedsUpdate || force ) {
        if ( this.parent ) {
          mat4.mul( this.matrixWorld,this.parent.matrixWorld,
this.modelMatrix );
        } else {
        mat4.copy(this.matrixWorld,this.modelMatrix );
      }
      this.matrixWorldNeedsUpdate = false;
      force = true;
    }
    // update children
    for ( var i = 0, len = this.children.length; i < len; i ++ ) {
      var child = this.children[ i ];
      if ( child instanceof Bone ) {
        child.update( this.identityMatrix, false );
      } else {
        child.updateMatrixWorld( true );
      }
    }
    // make a snapshot of the bones' rest position
    if ( this.boneInverses == undefined ) {
      this.boneInverses = [];
      for ( var b = 0, bl = this.bones.length; b < bl; b ++ ) {
        var inverse = mat4.create();
        mat4.invert(inverse, this.bones[ b ].skinMatrix );
        this.boneInverses.push( inverse );
      }
    }
    // flatten bone matrices to array
```

```
    for ( var b = 0, bl = this.bones.length; b < bl; b ++ ) {
      // compute the offset between the current and the original
transform
      // was already representing the offset; however, this
requires some
      // major changes to the animation system
      mat4.mul(offsetMatrix, this.bones[ b ].skinMatrix,
this.boneInverses[ b ] );
      this.flattenToArrayOffset(offsetMatrix, this.boneMatrices, b
* 16 );
    }
  };
}();
```

In the preceding code, `UpdateWorldMatrix` is the key function where all the computations happen. It performs the following steps:

1. It first computes its own `modelMatrix` and then updates its `matrixWorld` matrix (concatenates `parentMatrix`).
2. It iterates over the children array. If the child object is an instance of `bone`, then it invokes the `update` function to compute its `skinMatrix`. If the child object is an instance of `StageObject`, it invokes `updateWorldMatrix` of the child object to compute its `matrixWorld` matrix.

It then computes the offset matrix for each bone in the array. When the object of `RiggedMesh` is first initialized and `updateWorldMatrix` is invoked, it checks whether the `boneInverses` array is created. If they don't exist, it computes the inverse of each bone's `skinMatrix` and stores it in the `boneInverses` array. The bone's initial matrix is the binding matrix. This initial call stores the inverse binding matrix. Then in subsequent calls of `updateWorldMatrix`, it does not calculate the `boneInverses` array (discussed in the *The binding matrix* section of this chapter). The inverse binding matrix is only calculated once. It then computes $\texttt{offsetMatrix}_i=\texttt{B}^{-1}_i*\texttt{skinMatrix}$. While animating bones, `skinMatrix` will change each time, but the initial inverse binding matrix is calculated only once and stored in the `boneInverse` variable. When we load the `08-Loading-Skinned-Models.html` page in a browser, the character model maintains its starting pose. This happens because the initial skin matrix is multiplied by its own inverse. Hence the offset matrix for each bone becomes an identity matrix, so no vertex is transformed. During animation, a vertex is transformed from the skin local space to joint local space, and then we transform it using the joint's `skinMatrix`.

We iterate over the `bones` array and store the `offsetMatrices` values of each bone in a single flat array, `boneMatrices(this.flattenToArrayOffset(offsetMatrix, this.boneMatrices, b * 16 ))`. The flat array would look like:

```
boneMatrices[]=[bone11, bone12, bone13, bone14, bone21, bone22,
bone23, bone24, bone31, bone32, bon33, bone34, bone41, bone42,
bone43, bone44, bone111, bone112, bone113, bone114, bone121,
bone122, bone123, bone124, bone131, bone132, bone133, bone134,
bone141, bone142, bone143, bone144, bone211, bone212, bone213,
bone214, bone221, bone222, bone223, bone224, bone331, bone332,
bone333, bone334, bone341, bone342, bone343, bone344,...boneN11,
boneN12, boneN13, boneN14, boneN21, boneN22, boneN23, boneN24,
boneN31, boneN32, boneN33, boneN34, boneN41, boneN42, boneN43,
boneN44,]
```

The `flattenToArrayOffset` function takes `offsetMatrix` of each bone and the `boneMatrices` array and copies `offsetMatrix` at the location `b*16`. The `flattenToArrayOffset` function is defined as follows:

```
RiggedMesh.prototype.flattenToArrayOffset=function(mat, flat,
offset ) {
  var te = mat;
  flat[ offset ] = te[0];
  flat[ offset + 1 ] = te[1];
  flat[ offset + 2 ] = te[2];
  flat[ offset + 3 ] = te[3];
  flat[ offset + 4 ] = te[4];
  flat[ offset + 5 ] = te[5];
  flat[ offset + 6 ] = te[6];
  flat[ offset + 7 ] = te[7];
  flat[ offset + 8 ]  = te[8];
  flat[ offset + 9 ]  = te[9];
  flat[ offset + 10 ] = te[10];
  flat[ offset + 11 ] = te[11];
  flat[ offset + 12 ] = te[12];
  flat[ offset + 13 ] = te[13];
  flat[ offset + 14 ] = te[14];
  flat[ offset + 15 ] = te[15];
  return flat;
}
RiggedMesh.prototype.pose = function () {
  this.updateMatrixWorld( true );
  this.normalizeSkinWeights();
};
```

This `pose` function is invoked whenever the DOF value of any bone is modified. It is invoked from animation handlers and initially invoked from the `RiggedMesh` constructor. The `normalizeSkinWeights` function is defined as follows:

```
RiggedMesh.prototype.normalizeSkinWeights = function () {
  for ( var i = 0; i < this.geometry.skinIndices.length; i ++ ) {
    var sw = this.geometry.skinWeights[ i ];
    vec4.normalize(sw,sw);
  }
};
```

This another very important function makes sure that the sum of all weights for a vertex is equal to 1. If you open `model/obj/mrgreen.json`, you will notice `skinWeights=[1,1 …..]`. This simply means that vertex one in the `vertices` array is equally affected by both the bones in the `skinIndices` array. However, we need to normalize it before we pass them to the vertex shader. The `matrixWorld` matrix is updated as follows:

```
RiggedMesh.prototype.update=function(steps){
  this.updateMatrixWorld(true);
  this.updateMatrix();
}
```

The preceding `update` function simply updates the `matrixWorld`. The final step of cloning is done as follows:

```
RiggedMesh.prototype.clone = function ( object ) {
  if ( object === undefined ) {
    object = new RiggedMesh( this.geometry );
  }
  return object;
};
```

Loading the skinned model

Our base classes are ready. Now, let's move on to see them in action. We will need to modify our shaders to compute the vertex transformations. Open `08-Loading-Skinned-Models.html` in your favorite editor.

This vertex shader will now take four extra parameters, which are as follows:

- `useSkinning`: This is a `booluniform` value that determines whether the passed vertex uses skinning or not.
- `boneGlobalMatrices`: This is a mat4 `uniform` value that holds all the offset

matrices, passed as a flattened array from the main control code.

- `SkinIndex`: This is a `vec4attribute` value that holds the indices of the offset matrices (`boneGlobalMatrices`) affecting the vertex in question.
- `skinWeight`: This is a `vec4attribute` value that holds the weights of the corresponding bones in the `skinIndices` array affecting the vertex.

The vertex shader code for loading the skinned model is as follows:

```
<script id="shader-vs" type="x-shader/x-vertex">
  const int MAX_BONES = 100;
  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;
  uniform mat4 mVMatrix;
  uniform mat4 pMatrix;
  uniform mat4 nMatrix;
...
  uniform bool useSkinning;
  uniform mat4 boneGlobalMatrices[ MAX_BONES ];
  attribute vec4 skinIndex;
  attribute vec4 skinWeight;
  mat4 getBoneMatrix( const in float i ) {
  mat4 bone = boneGlobalMatrices[ int(i) ];
  return bone;
}
void main(void) {
...
  if(useSkinning) {
    mat4 boneMatX = getBoneMatrix( skinIndex.x );
    mat4 boneMatY = getBoneMatrix( skinIndex.y );
    vec4 skinVertex = vec4(aVertexPosition, 1.0);
    vec4 skinned  = boneMatX  skinVertex  skinWeight.x;
    skinned += boneMatY  skinVertex  skinWeight.y;
    skinned=mVMatrix skinned;
    vertexPos = skinned.xyz;
    gl_Position= pMatrixskinned;
    mat4 skinMatrix = skinWeight.x  boneMatX;
    skinMatrix  += skinWeight.y  boneMatY;
    vec4 skinnedNormal = skinMatrix  vec4( aVertexNormal, 0.0 );
    transformedNormal = vec3(nMatrix  skinnedNormal);
  }else{
    gl_Position= pMatrix *vertexPos4;
  }
...
}
</script>
```

The `getBoneMatrix` function takes the bone index as a parameter and returns the corresponding matrix, `(mat4 bone = boneGlobalMatrices[ int(i) ];)`.

In the code, we first check whether the vertex of a geometry uses skinning and get the X and Y bone matrices corresponding to the `skinIndex` variables, *x* and *y* (`mat4 boneMatX = getBoneMatrix( skinIndex.x )`). It then converts the vertex from `vec3` to `vec4` object and stores it in the `skinIndex` object (`vec4 skinVertex = vec4(aVertexPosition, 1.0)`). Then, it transforms the vertex with the corresponding matrices along with the weight (calculates the weighted average) and transforms the `skinned` vertex with `mvMatrix`.

```
vec4 skinned = boneMatX  skinVertex  skinWeight.x;
skinned += boneMatY  skinVertex  skinWeight.y;
skinned = mVMatrix * skinned;
gl_Position = pMatrix * skinned;
```

We then transform the normal using the same algorithm and multiply it with the normal vertex.

```
mat4 skinMatrix = skinWeight.x * boneMatX;
skinMatrix += skinWeight.y  boneMatY;
vec4 skinnedNormal = skinMatrix  vec4( aVertexNormal, 0.0 );
transformedNormal = vec3(nMatrix * skinnedNormal);
```

In a nutshell, we first transform the vertex and the normal with the bone offset matrix and follow our normal course of transforming them to the ModelView matrix (`mvMatrix`) and projection matrix (`pMatrix`).

The main control code for loading the skinned model has some functions explained as follows:

- The `start` function loads the JSON model with skinning and creates a `RiggedMesh` object instead of `StageObject` if the JSON object has the `bones` array defined in it.

  ```
  function start() {
  ...
    loadStageObject("model/obj/mrgreen.json",[600.0, 40.0, 0.0],
  0.0, 0.0, 0.0);
  ...
  }
  function loadStageObject(url, location, rotationX, rotationY,
  rotationZ) {
  ...
  ```

```
  else {
    if(data.bones) {
      stageObject=new RiggedMesh();
    }
    else{
      stageObject=new StageObject();
    }
  }
...
}
```

- The `initShaders` function accesses the attributes (`skinIndex` and `skinWeight`) and stores their references in the `shaderProgram` object.

```
function initShaders() {
...
  shaderProgram.skinIndex = gl.getAttribLocation(shaderProgram,
"skinIndex");
  shaderProgram.skinWeight =
gl.getAttribLocation(shaderProgram, "skinWeight");
  shaderProgram.useSkinning =
gl.getUniformLocation(shaderProgram, "useSkinning");
  shaderProgram.boneGlobalMatrices =
gl.getUniformLocation(shaderProgram, "boneGlobalMatrices");
...
}
```

- The `drawScene` function iterates over visible stage objects and checks whether the instance is of the `RiggedMesh` type. If it is valid, it enables the `skinIndex` and `skinWeight` attributes (`gl.enableVertexAttribArray(shaderProgram.skinIndex)`), activates the buffer objects (`gl.bindBuffer(gl.ARRAY_BUFFER, stage.stageObjects[i].skinIndexBuffer)`), and assigns their memory buffers to their corresponding attributes (`gl.vertexAttribPointer(shaderProgram.skinIndex, 4, gl.FLOAT, false, 0, 0)`). Then, it also assigns the offset bone matrices to the uniform `boneGlobalMatrices` (`gl.uniformMatrix4fv( shaderProgram.boneGlobalMatrices, false, stage.stageObjects[i].boneMatrices )`). Note that we assign `boneMatrices` as matrices of floats.

  If the object is not of the `RiggedMesh` type, we disable the attributes (`gl.disableVertexAttribArray(shaderProgram.skinIndex)`).

```
function drawScene() {
```

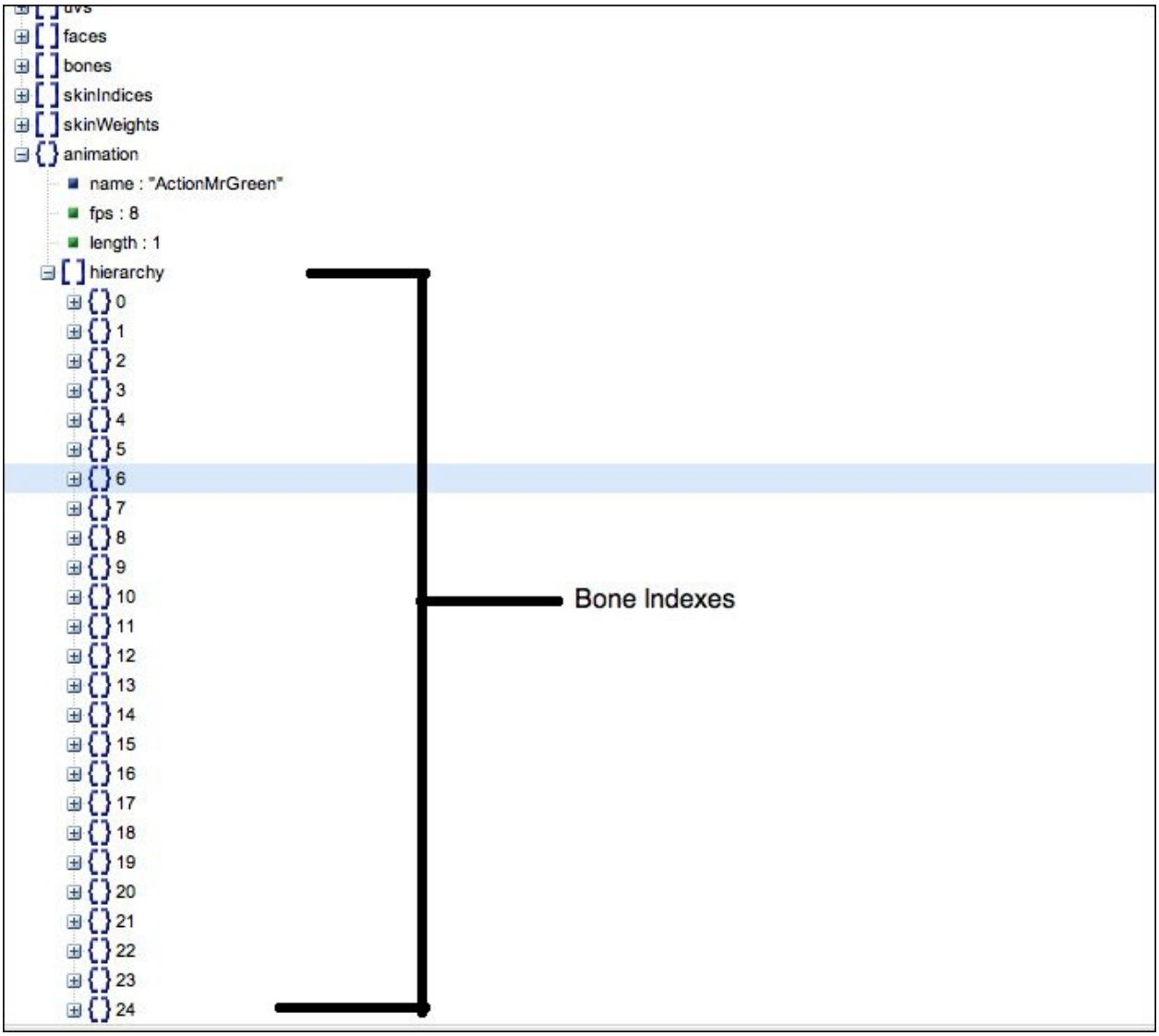
```
...
  if(stage.stageObjects[i] instanceof RiggedMesh) {
    gl.uniform1i(shaderProgram.useSkinning,1);
    gl.enableVertexAttribArray(shaderProgram.skinIndex);
    gl.enableVertexAttribArray(shaderProgram.skinWeight);

    gl.uniformMatrix4fv( shaderProgram.boneGlobalMatrices,
false, stage.stageObjects[i].boneMatrices );
    gl.bindBuffer(gl.ARRAY_BUFFER,
stage.stageObjects[i].skinIndexBuffer);
    gl.vertexAttribPointer(shaderProgram.skinIndex, 4,
gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER,
stage.stageObjects[i].skinWeightBuffer);
    gl.vertexAttribPointer(shaderProgram.skinWeight, 4,
gl.FLOAT, false, 0, 0);
  } else {
    gl.disableVertexAttribArray(shaderProgram.skinIndex);
    gl.disableVertexAttribArray(shaderProgram.skinWeight);

    gl.uniform1i(shaderProgram.useSkinning,0);
  }
...
}
```

Animating a rigged JSON model

Well, we did not take much effort to load a static model, rather we took more effort to animate the model. Hence, let's first understand how animation data is encoded in a JSON file.

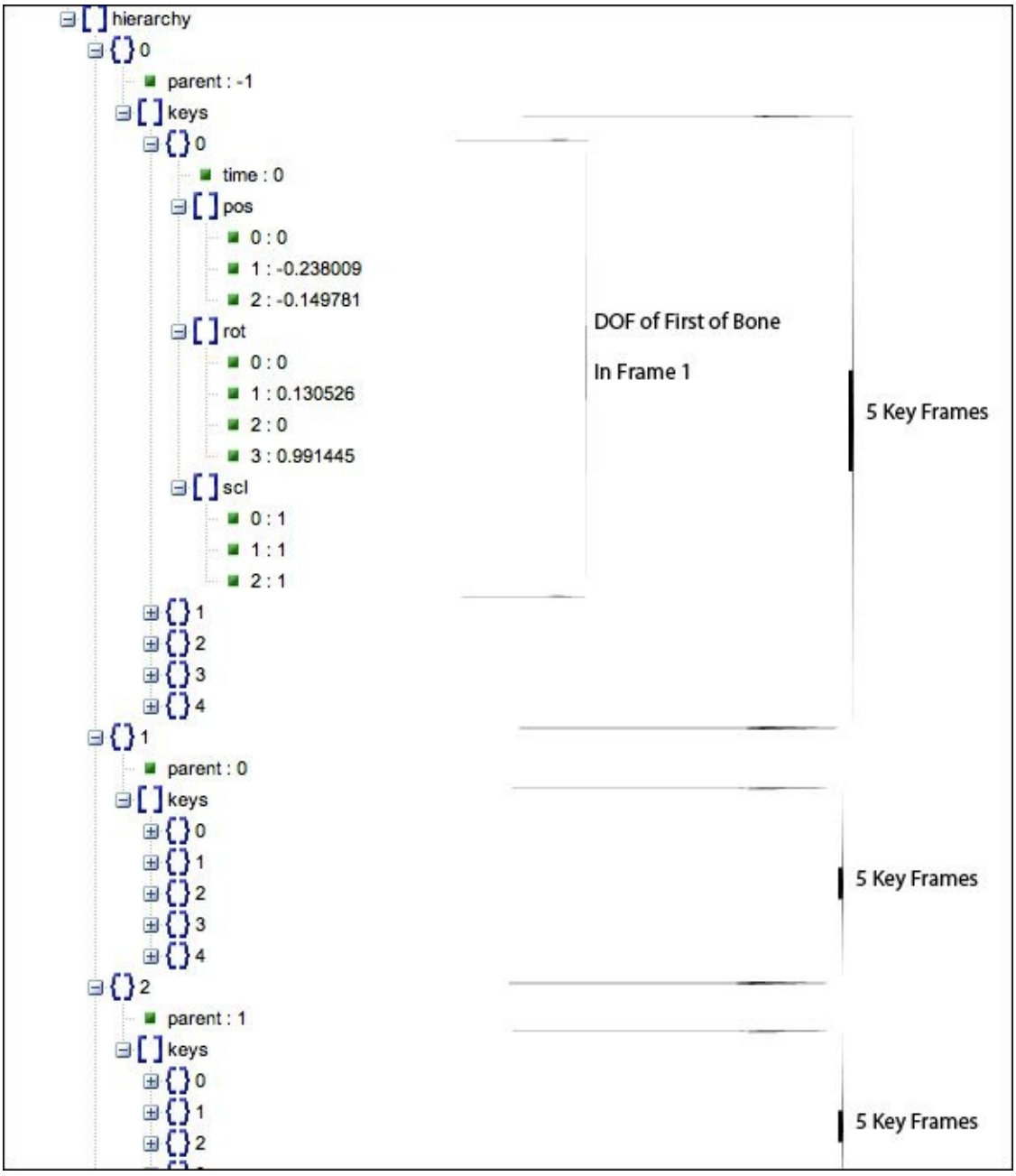
JSON model – animation data

Open `model/obj/mrgreen.json` in your text editor. In the file, you will find an element animation. When the file is viewed in any JSON viewer, you will see `name`, `fps`, `length`, and `hierarchy` as the animation object's child elements. To understand our JSON file, we used an online viewer (http://jsonviewer.stack.hu/). We loaded our JSON file in the viewer as shown in the following screenshot:



The other child elements are obvious but the most interesting element is `hierarchy`. The `hierarchy` element is an array; the length of this array is equal

to the length of bones. Each array element has animation data for the corresponding bone index. The `hierarchy` array element holds the keyframes (`keys`) element, which holds DOF (`position`, `rotation`, and `scale`) values for that frame, as shown in the following screenshot:



In the preceding screenshot, you will notice that the `hierarchy` element's first child (**0**) has a `parent` element (**-1**) and a `keys` array. The `keys` array has five

elements and each element has the `time`, `pos`, `rot`, and `scl` values. This shows that the animation has five frames. Similarly for all other bone elements, we will have the `keys` array with five elements. The `time` element defines the time at which the pose will be activated.

Loading the animation data

We took the basic strategy to load the animation data from `three.js` (`Animation` and `AnimationHandler`). We modified its code to handle our objects. So, let's understand the basic strategy. `AnimationHandler` is a static class and does three functions:

- Maintain the list of active objects of `Animation` to play.
- Parse the animation data and store it in an array. The index of the array is the name of animation.
- Invoke the `update` function of active objects of `Animation` when invoked with elapse time.

Open `primitive/AnimationHandler.js` in your editor. The `AnimationHandler` class maintains three arrays:

- `playing`: This array contains the list of animation objects
- `library`: This array maintains the animation data
- `that`: This array is the static object with functions.

The `update` function iterates over the list of `Animation` objects and invokes their `update` function with `deltaTimeMS`, the time since the last update call as shown in the following code snippet:

```
AnimationHandler = (function() {
  var playing = [];
  var library = {};
  var that    = {};
  //--- update ---
  that.update = function( deltaTimeMS ) {
    for( var i = 0; i < playing.length; i ++ )
      playing[ i ].update( deltaTimeMS );
  };
```

The `addToUpdate` and `removeFromUpdate` functions add and delete animation objects to and the from `playing` list, respectively, as shown in the following code snippet:

```
that.addToUpdate = function( animation ) {
    if ( playing.indexOf( animation ) === -1 )
      playing.push( animation );
  };
  that.removeFromUpdate = function( animation ) {
    var index = playing.indexOf( animation );
```

```
    var index = playing.indexOf( animation );

    if( index !== -1 )
      playing.splice( index, 1 );
  };
```

The `add` function pushes the animation data on to the `library` array, with the key as the name of the animation data. It then invokes `initData`. The `get` function returns the animation data from the array. The `add` function is defined as follows:

```
that.add = function( data ) {
    library[ data.name ] = data;
    initData( data );

  };
  that.get = function( name ) {
    if ( typeof name === "string" ) {
      if ( library[ name ] ) {
        return library[ name ];
      } else {
        return null;
      }
    }
  };
```

The `parse` function pushes all the bones of the `RiggedMesh` class to the `hierarchy` array. This function is invoked from the `Animation` class. The `Animation` class maintains the list of bones of the object. It updates the bones' properties (`position`, `rotation`, and `scale`) in its `update` function. The `parse` function is defined as follows:

```
that.parse = function( root ) {
    // setup hierarchy
    var hierarchy = [];
    if ( root instanceof RiggedMesh ) {
      for( var b = 0; b < root.bones.length; b++ ) {
        hierarchy.push( root.bones[ b ] );
      }
    }
    return hierarchy;
  };
```

The `initData` function parses the animation object from the JSON file. The `hierarchy` object simply maintains the key frames (`time`, `position`, `scale`, and

`rotation`) for each bone, eliminates any repeated animation data (key frames at the same time), and also updates the time to 0 if the time of the key frame is negative. Basically, it sanitizes the data and sees that it is not processed again using the `initialized` property.

```
var initData = function( data ) {

    if( data.initialized === true )
      return;
    for( var h = 0; h < data.hierarchy.length; h ++ ) {
      for( var k = 0; k < data.hierarchy[ h ].keys.length;k ++ ) {
        // remove minus times
        if( data.hierarchy[ h ].keys[ k ].time < 0 )
          data.hierarchy[ h ].keys[ k ].time = 0;
        // create quaternions
        var quater = data.hierarchy[ h ].keys[ k ].rot;
        data.hierarchy[ h ].keys[ k ].rot = quat.fromValues(
quater[0], quater[1], quater[2], quater[3] );
      }
      for ( var k = 1; k < data.hierarchy[ h ].keys.length;k ++ ) {
        if ( data.hierarchy[ h ].keys[ k ].time === data.hierarchy[
h ].keys[ k - 1 ].time ) {
          data.hierarchy[ h ].keys.splice( k, 1 );
          k --;
        }
      }
      for ( var k = 0; k < data.hierarchy[ h ].keys.length;k ++ ) {
        data.hierarchy[ h ].keys[ k ].index = k;
      }
    }
    data.initialized = true;
  };
  return that;
}());
```

Open `primitive/Animation.js` in your text editor.

The `Animation` class follows a very basic strategy. It maintains the previous and next key frame objects, takes the elapsed time, gets the frame in which the elapsed time falls, and then interpolates between the values of the frame using the elapsed time. For example, if the elapsed time is 0.5, frame two has time stamp 0.4, frame three has time stamp 0.6, the value of pos.x equals 10 in frame two and value of pos.x equals 20 in frame three; then, it sets the bone's position using the formula:

scale=(currentTime-prevKeyTime)/(nextKeytime-prevKeyTime)

pos.x=prevKey.x+(nextKey.x-prevKey.x)*scale

=~10+(20-10)*(0.5-.4)/(0.6-0.4)

The preceding code is an example of linear interpolation as the value of *x* is a direct function of time (scale).

The constructor of `Animation` takes two parameters, the name of the `RiggedMesh` object and the name of the animation data to play. It then retrieves the bone hierarchy using the `parse` function of `AnimationHandler` as well as the animation data using its `get` function.

```
Animation = function ( root, name) {
  this.root = root;
  this.data = AnimationHandler.get( name );
  this.hierarchy = AnimationHandler.parse( root );
  this.currentTime = 0;
  this.timeScale = 1;
  this.isPlaying = false;
  this.isPaused = true;
  this.loop = true;
};
```

The `play` function initializes parameters such as `isPlaying` to `true` and `isPaused` to `false`. It creates the animation cache that basically holds the `nextKey` and `preyKey` values for all bones in the mesh. It iterates over all bones and if the animation cache is not defined for the bone, it gets the data (`pos`, `rot`, and `scale`) from the first key (`prevKey.pos = this.data.hierarchy[ h ].keys[ 0 ];`) and the second key (`nextKey.pos = this.getNextKeyWith( "pos", h, 1 );`) from the animation data and stores its `prevKey` and `nextKey` variables. It then adds the `Animation` object itself to the `playing` list of the animation handler (`AnimationHandler.addToUpdate( this );`) as shown in the following code snippet:

```
Animation.prototype.play = function ( loop, startTimeMS ) {
  if ( this.isPlaying === false ) {

    this.isPlaying = true;
    this.loop = loop !== undefined ? loop : true;
    this.currentTime = startTimeMS !== undefined ? startTimeMS : 0;
    // reset key cache
    var h, hl = this.hierarchy.length, object;
```

```
    var h, hl = this.hierarchy.length,object;
    for ( h = 0; h < hl; h ++ ) {
      object = this.hierarchy[ h ];
      object.matrixAutoUpdate = true;
      if ( object.animationCache === undefined ) {
        object.animationCache = {};
        object.animationCache.prevKey = { pos: 0, rot: 0, scl: 0 };
        object.animationCache.nextKey = { pos: 0, rot: 0, scl: 0 };
        object.animationCache.originalMatrix = object instanceof
Bone ? object.skinMatrix : object.matrix;
      }
      var prevKey = object.animationCache.prevKey;
      var nextKey = object.animationCache.nextKey;
      prevKey.pos = this.data.hierarchy[ h ].keys[ 0 ];
      prevKey.rot = this.data.hierarchy[ h ].keys[ 0 ];
      prevKey.scl = this.data.hierarchy[ h ].keys[ 0 ];
      nextKey.pos = this.getNextKeyWith( "pos", h, 1 );
      nextKey.rot = this.getNextKeyWith( "rot", h, 1 );
      nextKey.scl = this.getNextKeyWith( "scl", h, 1 );
    }
    this.update( 0 );
  }
  this.isPaused = false;
  AnimationHandler.addToUpdate( this );
};
```

The `update` function is the heart of the animation system. The `types` array holds DOF names (`pos`, `rot`, and `scl`), which are indexed to get the key data. The `update` function takes the elapse time (`deltaTimeMS`) as a parameter. It is invoked from the `AnimationHandler` class's `update` function.

```
Animation.prototype.update = function ( deltaTimeMS ) {
  if ( this.isPlaying === false ) return;
  var types = [ "pos", "rot", "scl" ];
  var type;
  var scale;
  var vector;
  var prevXYZ, nextXYZ;
  var prevKey, nextKey;
  var object;
  var animationCache;
  var frame;
  var currentTime, unloopedCurrentTime;
  var currentPoint, forwardPoint, angle;
```

We first calculate the current time by adding the elapsed time to it as shown in the following code snippet:

the following code snippet:

```
this.currentTime += deltaTimeMS;
  unloopedCurrentTime = this.currentTime;
```

The `unloopedCurrentTime` value is maintained to see if the current time does not exceed the total animation time. If it is not a looping animation, then this variable is checked to stop the animation. Then, we find the moduli of the current time to eliminate the natural number from the current time.

currentTime=0.9;

currentTime(0.9) + elapsedTime(0.2) = unloopedCurrentTime(1.1);

currentTime(1.1)%1 = currentTime(0.1).

The `unloopedCurrentTime` value becomes greater than `1`. We remove the natural number from the `currentTime`. At that time, the `unloopedCurrentTime` is not equal to `currentTime`.

```
currentTime = this.currentTime = this.currentTime %
this.data.length;
```

We then iterate over all bones as follows:

```
for ( var h = 0, hl = this.hierarchy.length; h < hl; h ++ ) {
    object = this.hierarchy[ h ];
    animationCache = object.animationCache;
```

Iterate over types to get the key information for types (`pos`, `rot`, and `scl`) shown as follows:

```
for ( var t = 0; t < 3; t ++ ) {
```

Get the current, previous, and next key frames for the bone and `type` from the animation cache.

```
type    = types[ t ];
      prevKey = animationCache.prevKey[ type ];
      nextKey = animationCache.nextKey[ type ];
```

Check if the next frame's time is less than `unloopedCurrentTime`. This means it has to pick the next frame, otherwise use the same frame to set the DOF of the bone.

```
if ( nextKey.time <= unloopedCurrentTime ) {
```

The `currentTime` value will always be equal to `unloopedCurrentTime`, but when the animation has run its complete cycle once, then the `unloopedCurrentTime` value will be greater than one and `currentTime` will be less than it. This means the animation was run once. So, we have to pick the next frame only when `this.loop` is valid, otherwise we have to stop the animation.

```
if ( currentTime < unloopedCurrentTime ) {
```

If the loop is valid, then pick the first frame and second frame to set the `prevKey` and `nextKey` variables and loop until we do not find the correct `nextKey`, whose time is less than the current time. Stop the animation if looping is not enabled.

```
if ( this.loop ) {
            prevKey = this.data.hierarchy[ h ].keys[ 0 ];
            nextKey = this.getNextKeyWith( type, h, 1 );
            while( nextKey.time < currentTime ) {
              prevKey = nextKey;
              nextKey = this.getNextKeyWith( type, h, nextKey.index
+ 1 );
            }
          } else {
            this.stop();
            return;
          }
        } else {
```

If `currentTime` equals `unloopedCurrentTime`, then the animation has not completed once. Simply iterate over all keys until the value of `currentTime` is less than that of `nextKey`. This will give us our new next and previous keys as shown in the following code snippet:

```
do {
            prevKey = nextKey;
            nextKey = this.getNextKeyWith( type, h, nextKey.index +
1 );
          } while( nextKey.time < currentTime )
        }
```

Set the new `prevKey` and `nextKey` values to the animation cache.

```
animationCache.prevKey[ type ] = prevKey;
        animationCache.nextKey[ type ] = nextKey;
      }
```

Set the object's `matrixWorldNeedsUpdate` value to `true` so that `updateWorldMatrix` of the bone calculates our new `skinMatrix`.

```
object.matrixAutoUpdate = true;
      object.matrixWorldNeedsUpdate = true;
```

Calculate `scale` to interpolate between the `prevKey` and `nextKey` values.

```
scale = ( currentTime - prevKey.time ) / ( nextKey.time -
prevKey.time );
      prevXYZ = prevKey[ type ];
      nextXYZ = nextKey[ type ];
      if ( scale < 0 || scale > 1 ) {
        scale = scale < 0 ? 0 : 1;
      }
```

If the `type` value equals position, calculate the new position between the `preyKey` and `nextKey` values and update the bones' position vector with the new interpolated value. Thus, we interpolate based on the value of `scale`. We use spherical linear interpolation to calculate the new quaternion between `prevKey` and `nextKey`.

```
if ( type === "pos" ) {
        vector = object.position;
        vector.x = prevXYZ[ 0 ] + ( nextXYZ[ 0 ] - prevXYZ[ 0 ] )
scale;
        vector.y = prevXYZ[ 1 ] + ( nextXYZ[ 1 ] - prevXYZ[ 1 ] )
scale;
        vector.z = prevXYZ[ 2 ] + ( nextXYZ[ 2 ] - prevXYZ[ 2 ] ) *
scale;

      } else if ( type === "rot" ) {
        quat.slerp(object.quaternion, prevXYZ, nextXYZ,  scale );
      } else if ( type === "scl" ) {
        vector = object.scale;
        vector.x = prevXYZ[ 0 ] + ( nextXYZ[ 0 ] - prevXYZ[ 0 ] )
scale;
        vector.y = prevXYZ[ 1 ] + ( nextXYZ[ 1 ] - prevXYZ[ 1 ] )
scale;
        vector.z = prevXYZ[ 2 ] + ( nextXYZ[ 2 ] - prevXYZ[ 2 ] ) *
scale;
      }
    }
  }
};
```

The `getNextKeyWith` function is used extensively by the `update` function. It basically takes the bone index, `type` (`pos`, `rot`, and `scl`), and the `keys` array's index to get the next frame.

```
Animation.prototype.getNextKeyWith = function ( type, h, key ) {
  var keys = this.data.hierarchy[ h ].keys;
  key = key % keys.length;
  for ( ; key < keys.length; key++ ) {
    if ( keys[ key ][ type ] !== undefined ) {
      return keys[ key ];
    }
  }
  return this.data.hierarchy[ h ].keys[ 0 ];
};
```

Open `08-Loading-Skinned-Animatons.html` to learn how to use the preceding classes.

After we initialize our `RiggedMesh` object, we check whether the name of the `RiggedMesh` object is `mrgreen`. We initialize our `Animation` object with the stage object and the name of animation to play. We also add the animation data from the `geometry` to the `AnimationHandler`.

```
function loadStageObject(url, location, rotationX, rotationY,
rotationZ) {
...
  if(stageObject.name=="mrgreen") {
    AnimationHandler.add( stageObject.geometry.animation );
    animation = new Animation( stageObject, "ActionMrGreen" );
    animation.play( true );
    animation.update( 0 );
  }
...
}
```

The `animate` function on main code invokes the `AnimationHandler` class' `update` function. It passes the elapsed time as a parameter to the function. `AnimationHandler` class's `update` function in turn invokes the `update` function of `RiggedMesh`.

Hence, now our `animate` function does three things: updates the physical world, updates `AnimationHandler`, and then redraws the scene.

```
function animate() {
  currentTime = (new Date).getTime();
```

```
  currentTime = (new Date).getTime();
  elapsedTime = currentTime - lastFrameTime;
  if (elapsedTime < rate) return;
  AnimationHandler.update(elapsedTime/1000);
...
}
```

If you have your `RiggedMesh` class in place, animation data is simple to understand and work with.

Exporting models from 3D software in JSON

In our code, we strictly follow the three.js JSON 3.1 format (https://github.com/mrdoob/three.js/wiki/JSON-Model-format-3.1). Hence, we can directly use `three.js` exporters to load 3D objects in our code.

Note

Our code only handles models with a single material/texture file. Hence, if you use the code directly, make sure that you create models with a single texture.

Exporting from Blender

We can download the Blender add-on from here https://github.com/mrdoob/three.js/tree/master/utils/exporters/blender/2.65. We can install it in Blender using following simple steps:

1. Copy the add-on to the following path:
    - Windows: `C:\Users\USERNAME\AppData\Roaming\Blender Foundation\Blender\2.6X\scripts\addons`
    - Mac: *`Applications`*`Blender/blender.app/Contents/MacOS/2.6X/scripts/`
    - Linux: *`home`*`USERNAME/.config/blender/2.6X/scripts/addons`
2. In Blender preferences, in add-ons, look for option three.
3. Enable the checkbox next to **Import-Export: three.js format**.

Now, you can use the **Export** menu to export the JSON file.

Converting FBX/Collada/3DS files to JSON

The conversion utility is placed in the `code/model/convert_obj_three.py` folder. You can simply fire the following command at the terminal or the DOS window:

```
python convert_obj_three.py sample.fbx sample.js -t
```

To use this utility, you must have Python installed on your system along with FBX Python bindings (http://usa.autodesk.com/fbx/).

Also, note that the utility does not export animation data.

Loading MD5Mesh and MD5Anim files

MD5Mesh and MD5Anim are very popular 3D formats. They are lightweight and exporters are available for all major software like Blender, Maya, and 3ds Max.

We download a utility to convert MD5 files to the JSON format using JavaScript from http://oos.moxiecode.com/js_webgl/md5_converter/. We also modified its code and have added it to our `code/js` folder (`code/js/MD5_converter.js`). You may simply include this file in your code and use the following sample code to load the MD5 files:

```
function loadMD5() {
  var meshLoaded=false;
  var animLoaded=false;
  var mesh="";
  var anim="";
  $.get("model/md5/boblampclean.md5mesh",function(data) {
    meshLoaded=true;
    mesh=data;
    if(meshLoaded&&animLoaded) {
      processMD5(mesh,anim);
    }
  });
  $.get("model/md5/boblampclean.md5anim",function(data) {
    animLoaded=true;
    anim=data;
    if(meshLoaded&&animLoaded) {
      processMD5(mesh,anim);
    }
  });
}
```

Once both the mesh and animation files are loaded, we invoke `processMD5`. The function basically invokes `MD5_converter.process_md5anim` and `MD5_converter.process_md5mesh` and stores the JSON string in `meshData.string` and the animation string in `meshData.stringAnimationOnly`. Then, we parse the string to convert it to a JSON object using the jQuery function, `JSON.parse`. Then, we create a `RiggedMesh` object and invoke its `loadObject` function to load the JSON model.

```
function processMD5(mesh,animation) {
  var animData = MD5_converter.process_md5anim( animation,
"boblampclean.md5anim" );
```

```
  boblampclean.md5anim" );
  var meshData = MD5_converter.process_md5mesh( mesh,
"boblampclean.md5mesh" );
  var model=JSON.parse(meshData.string);
  var animationData=JSON.parse(meshData.stringAnimationOnly);
  model.animation=animationData.animation;
  model.metadata.sourceFile="boblampclean.md5mesh";
  var stageObject=new RiggedMesh();
  stageObject.loadObject(model);
  loadTexture(stageObject,false);
  addStageObject(stageObject,[0.0,40.0,-50.0],0.0,0.0,0.0);
}
```

Summary

This chapter was dedicated to loading and using skinned models. We covered the basics to help us understand how skinning data is stored in a JSON model file. We also explained the relation between the bones and the vertex transformations and used this knowledge to write our vertex shader. We implemented the smooth skinning algorithm to store our joint and skin data.

We then learned how animation data is encrypted in a JSON file. We evolved our classes to load the animation data and learned how to animate a model on frame-based animation principles.

This chapter used matrix transformations, quaternion mathematics and linear interpolation to handle difficult concepts such as animating skinned models.

In the next chapter, we will cover concepts such as picking and post-processing effects. Picking is the core concept used in all 3D editing software. Through picking, we can select 3D objects on a 2D screen.

Post-processing helps us add effects to a scene. For example, we may want our game user to feel as if he is viewing our game scene through night vision goggles.

Chapter 9. Ray Casting and Filters

Collision detection is a powerful tool to calculate hits and the intersection of two or more objects in a game scene but not sufficient for most cases where we need to predict collision before it actually happens. This chapter will unveil a very powerful concept in game development called **ray casting**, to cover cases that cannot be handled by collision detection. We will use ray casting to implement another very interesting technique called **picking**, which is used by CAD software. Picking is useful for many interactive use cases, where we need to map a mouse click by the user (2D screen coordinates) to the local/world space of the objects in the scene.

Framebuffers is another very powerful concept used in game development. It has many useful use cases from the implementation of view caching to implementations of filters. We will cover the following topics in this chapter:

- Basic concepts of ray casting
- The basics of picking
- Implementing picking using ray casting
- Framebuffers
- Implementing filters using framebuffers

Understanding the basic ray casting concepts

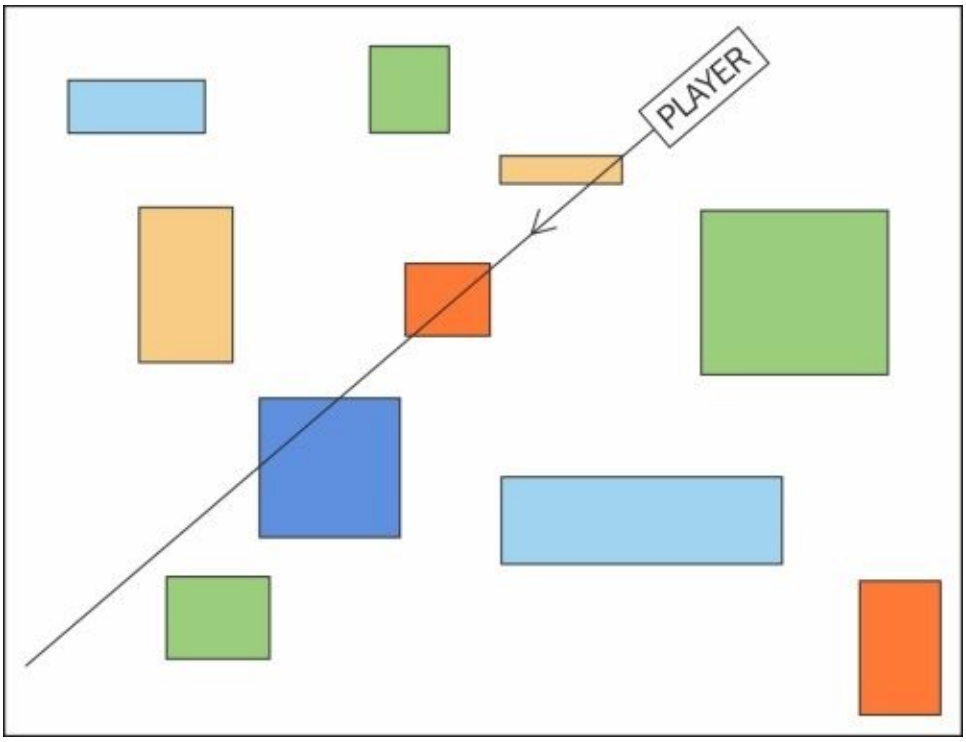
There are many use cases where collision detection is detected too late in a game. Let's consider a case, in a game, where we have a door and if the player approaches it, you want the door to open automatically. If we use collision detection for this case, then the player has to first collide with the door, in order to open it; or alternatively, you can create a bigger bounding box (a rigid body with more depth) so that when the user approaches the door a collision is detected earlier. However, this approach does not cover the use case of the player not facing the door. For instance, if the player has its back towards the door, then a collision would be detected and the door would open. Collision detection does not consider directions.

Ray casting is also used to build the basic AI of a non-playing character. A moving player might need to alter its direction or speed automatically when there is another existing object moving in the same direction. Collision is detected only after two objects have intersected, and often, we need to predict it. In such cases, ray casting is an effective concept to predict a collision.

Ray casting is used in many cases like:

- Determining the first object intersection by a ray
- Volume ray casting in a direct volume rendering method
- For hidden surface removal in finding the first intersection of a ray cast from the camera through each pixel of the image

In ray casting, geometric rays are traced from the position of the player to sample a light ray traveling in the direction of the player. If the traveling ray intersects any colliders (rigid bodies in our case), a hit is observed, and then we can take the desired action. The following diagram shows a simple ray casted from a player's position and its intersection with objects:



We will use the JigLib library to understand the implementation of a ray cast. JigLib provides us with two classes to implement ray casting: `JRay` and `JSegment`. The constructor of the `JRay` class is defined as follows:

```
JRay=function(_origin, _dir)
```

The constructor of this class takes two parameters, `_origin` and `_dir`. We will set the `_origin` parameter to the position of the character and the `_dir` parameter to the direction of the character. The position and direction can be easily derived from our `matrixWorld` matrix. The position will be derived from the *m30*, *m31*, and *m32* components of our matrix and the direction vector can be calculated by multiplying the forward vector with the rotation matrix (derived from `this.quaternion` of our `StageObject` class) or retrieving the *m03*, *m13*, and *m23* components of our `matrixWorld` matrix. The `JRay` class is used internally by the `JSegment` class. We will only interact with the `JSegment` class. The constructor of the `JSegment` class is defined as follows:

```
JSegment=function(_origin, _delta)
```

The `JSegment` class is similar to the `JRay` class except that it takes the `_delta`

parameter instead of the `_dir` parameter. The `_delta` parameter is the length of the ray multiplied by the direction of the player. The `JSegment` class does not cast an infinite ray, and the intersecting objects in the vicinity of the ray are returned. Let's quickly take a peep into the implementation:

```
vec3.scale(directionVector,ray.directionVector,100)
var segment=new jigLib.JSegment(origin,directionVector);
var out={};
var cs=system.getCollisionSystem();
if(cs.segmentIntersect(out, segment, null)){
  returnout.rigidBody;
}
```

First, we scale our direction vector by `100`, which is the length of our ray in the example. Then, we create an object of the segment. We retrieve the object of our collision system from the physics system object. We then check for an intersecting object using the `cs.segmentIntersect(out, segment, null)` function. The last parameter (`null`) is for the outer body, the enclosing body, not necessarily the parent of the object. It can be a skybox or any similar object, which we will ignore for our calculations. In our case, we do not pass the enclosing body; hence, we pass null. If the function returns true, then the intersecting object is retrieved from the `out` variable, which we passed as the first parameter in the function call.

The returned `out` parameter holds information such as `out.position`, `out.rigidbody`, and `out.frac`:

- `out.position`: This variable holds the position of the collider at the time of intersection with the ray, in case the collider is moving.
- `out.rigidBody`: This variable holds the collider the ray intersected with.
- `out.frac`: This variable holds the probability of intersection of the object with the ray. This is used internally by the function to check the most probable object intersecting with the ray.

Learning the basics of picking

All 3D platforms have a very basic problem to solve: what is under the mouse? This problem in the 3D world is referred to as picking. It is not simple and straightforward and involves nontrivial mathematics and some complex algorithms, but it is solvable once you understand the core concept. To solve the problem, there are two approaches in picking, based on an object color and ray casting.

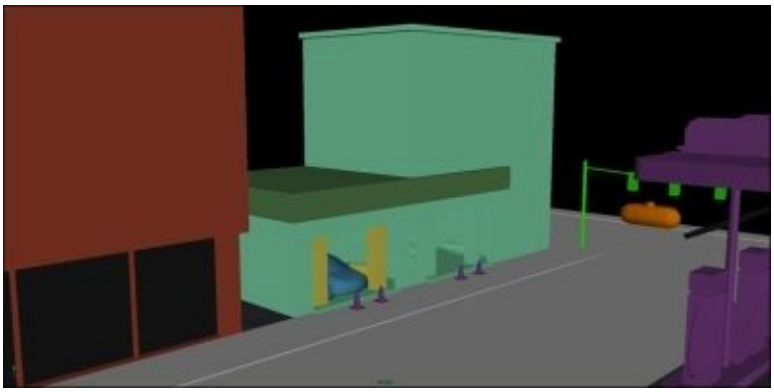
Picking based on an object's color

Each object in the scene is assigned a unique diffuse color, and then the complete scene is rendered in an offscreen framebuffer. When the user clicks on the scene, we get the color from the texture in a framebuffer of the corresponding click coordinate. Then, we iterate through the list of objects and match the retrieved color with the object diffuse color. Let's look at the pseudo code:

```
createFrameBuffer(framebuffer);
//Function invoked at each tick;
function render(){
  assignDiffuseColorToEachObject();
  gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
  drawScene();
  assignActualTextureOrColorToEachObect();
  gl.bindFramebuffer(gl.FRAMEBUFFER, null);
  drawScene();
}
```

If you observe the preceding pseudo code, you will realize that we render the scene twice; once in the offscreen framebuffer with the assigned color code for each object and then on the onscreen buffer with the actual texture and the color data. The following two screenshots explain the concept; the first screenshot shows how the scene is rendered in the framebuffer and the second screenshot shows how the scene appears on the actual screen. When the user clicks on the scene, click coordinates are mapped to the offscreen buffer texture.

The disadvantage is that although the implementation is straightforward, it requires double rendering of the same scene, which makes the overall rendering process slow. Also, you cannot select a single triangle from a mesh. We can only select complete objects. The following screenshot is rendered in the framebuffer, each object with a unique color:

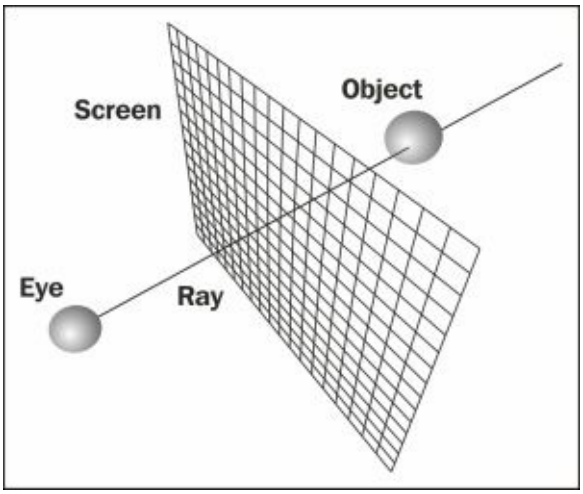


The following screenshot is rendered on the screen, each object with its actual texture:

Although we will discuss framebuffers in the *Rendering offscreen using framebuffers* section, we will not implement picking using the preceding technique in this book.

Picking using ray casting

We have discussed ray casting in the previous section, *Understanding the basic ray casting concepts*. In that section, we discussed that a ray was generated from a player's position and direction to detect any object's intersection. Now, let's replace the player position with screen coordinates where the mouse was clicked and derive the direction from the camera direction (also called the eye vector). Hence, picking using ray casting is implemented by casting a ray from the camera's position and direction into the scene—the objects intersected by the ray are the user selected objects. This is shown in the following diagram:



Let's understand the power of this technique. Using this technique, we can select objects up to any level of detail, but there is always a tradeoff in precision versus performance. So, let's understand the possibilities:

- For `jigLib.JBox` colliders, if a scene object is associated with the `JBox` or `JSphere` collider, then you will only be able to select a complete object. The `Jbox`, `JSphere`, and `JCapsule` colliders are also known as bounding box colliders. They are the fastest colliders. They have minimum computation load as the exact shape of the 3D object is not considered but a simple geometry is checked against the intersection. We have already associated these colliders with our scene objects in Chapter 7, *Physics and Terrains*.
- For `jigLib.JTriangleMesh` colliders, if a scene object is associated with a `JTriangleMesh` collider, then you can select a triangle/polygon in a scene object. Let's look at the constructor of `JTriangleMesh`:

```
JTriangleMesh=function(skin, maxTrianglesPerCell, minCellSize)
this.jigLibObj=new jigLib.JTriangleMesh(null, 100, 0.1);
this.jigLibObj.createMesh(triangles.verts, triangles.faces);
```

The constructor of the `JTriangleMesh` collider takes a hint of the size of an object by taking `maxTrianglesPerCell` and `mincellSize` parameters. This class has a function called `createMesh` which takes `triangles.verts` and `triangles.faces` as parameters. These parameters refer to the vertices and indices in the mesh. So basically, a `JTriangleMesh` collider knows the exact shape and size of the scene object, but it is very slow. It basically checks each triangle in the mesh for collision. Even though it is precise, it certainly is computationally expensive. However, we can use the `JTriangleMesh.prototype.segmentIntersect=function(out, seg, state)` function of the class to get the exact triangle that our ray/segment intersected. This method of finding an intersection is precise but slow.

We can use any type of collider in our scene for picking, but remember that we have to choose between precision and performance.

Implementing picking using ray casting

Let's now start implementing what we have discussed. The algorithm we will apply is as follows:

1. Create a rigid body for each scene object.
2. Calculate the click coordinates of the canvas, convert these coordinates to normalized device coordinates, and create a vector.
3. "Unproject" the vector using the camera projection matrix.
4. Create a ray segment using the vector.
5. Check for an intersection.
6. If an intersection is found, then change the color of the selected object.

Using a rigid body (collider) for each scene object

Now, we will modify our `StageObject` class again to create a bounding box. If we revisit the constructor of our `jigLib.JBox` class, then its constructor takes the width, depth, and height of the corresponding 3D model as follows:

```
var JBox=function(skin, width, depth, height)
```

When we add our 3D model to the stage, we do not know its height, width, and depth. Hence, we first need to create its bounding box in order to compute the preceding values.

A bounding box consists of the minimum and maximum values of the components (*x*, *y*, *z*) of the vertices of a mesh. Note that it is not the minimum and maximum vertices but the minimum/maximum values of the components of a vertex. Open the `StageObject.js` file from the `primitive` folder in your editor. The following code snippet is present in this file:

```
StageObject.prototype.calculateBoundingBox=function (){
  if(this.geometry.vertices.length>0){
    var
point=vec3.fromValues(this.geometry.vertices[0],this.geometry.verti
ces[1],this.geometry.vertices[2]);
    vec3.copy(this.min, point );
    vec3.copy(this.max, point );
    for ( var i = 3; i<this.geometry.vertices.length;  i=i+3) {

point=vec3.fromValues(this.geometry.vertices[i],this.geometry.verti
ces[i+1],this.geometry.vertices[i+2])
      if ( point[0] <this.min[0] ) {
        this.min[0] = point[0];
      } else if ( point[0] >this.max[0] ) {
        this.max[0] = point[0];
      }
      if ( point[1] <this.min[1] ) {
        this.min[1] = point[1];
      } else if ( point[1] >this.max[1] ) {
        this.max[1] = point[1];
      }

      if ( point[2] <this.min[2] ) {
        this.min[2] = point[2];
      } else if ( point[2] >this.max[2] ) {
        this.max[2] = point[2];
      }
```

```
      }
    }
  }
}
```

In the preceding code, we first store the first vertex in the `min` and `max` class variables of the `StageObject` class. We then iterate over the list of vertices of our geometry, compare each component (*x*, *y*, *z*) of a vertex with the `min` and `max` components, and store the minimum and maximum values in the corresponding components. You must have noticed that the `min` and `max` vectors are not actual vertices in the mesh, but they just hold the minimum and maximum values of *x*, *y*, and *z*.

The width of the 3D model is calculated using the following formula. Here, *max[0]* is the maximum value of *x* and *min[0]* is the minimum value of *x*:

Width = max[0] - min[0]

The height of the 3D model is calculated using the following formula. Here, *max[1]* is the maximum value of *y* and *min[1]* is the minimum value of *y*:

Height = max[1] - min[1]

The depth of the 3D model is calculated using the following formula. Here, *max[2]* is the maximum value of *z* and *min[2]* is the minimum value of *z*:

Depth = max[2] - min[2]

We create a collider which closely resembles the shape and real world physics of the 3D model. For example, `JCapsule` colliders are used for cylindrical objects which roll in a direction. For example, we have a ball, so we would not want its collider to be a box because a box collider would not roll. Hence, we also create a bounding sphere if the object resembles a sphere. The constructor of `jigLib.JSphere=function(skin, r)` takes the radius as an argument. Hence, we first need to compute its center to compute its radius. In the following code, we first compute the bounding box to calculate the center:

```
StageObject.prototype.calculateBoundingSphere=function (){
  this.calculateBoundingBox();
  this.center=this.calculateCenter();
  var squaredRadius=this.radius*this.radius;
  for ( var i = 0; i<this.geometry.vertices.length;  i=i+3) {
    var
```

```
    var
point=vec3.fromValues(this.geometry.vertices[i],this.geometry.verti
ces[i+1],this.geometry.vertices[i+2]);

squaredRadius=Math.max(squaredRadius,vec3.squaredDistance(this.cent
er,point));
  }
  this.radius=Math.sqrt(squaredRadius);
}
StageObject.prototype.calculateCenter=function (){
  var center=vec3.create();
  vec3.add(center,this.min,this.max);
  vec3.scale(center,center,0.5);
  return center;
}
```

Then, we compute the center of the sphere. The computation of the center is simple:

center = (min + max)/2

The bounding box gives us the `min` and `max` values, and we calculate the sphere's center from it. Then, we iterate over the list of vertices to find the distance of each vertex from the center and store the maximum radius using the `squaredRadius=Math.max(squaredRadius, vec3.squaredDistance(this.center, point));` statement.

Now, as our code to compute our bounding box is complete, we now need to initialize our rigid body using the `initializePhysics` function as follows:

```
StageObject.prototype.initializePhysics=function(sphere){
  if(sphere){
    this.calculateBoundingSphere();
    this.rigidBody= new jigLib.JSphere(null,this.radius);
    this.rigidBody.set_mass(this.radius*this.radius*this.radius);
  }else{
    this.calculateBoundingBox();
    var subVector=vec3.create();
    vec3.sub(subVector,this.max,this.min);
    this.rigidBody=new
jigLib.JBox(null,Math.abs(subVector[0]),Math.abs(subVector[2]),Math
.abs(subVector[1]));

this.rigidBody.set_mass(Math.abs(subVector[0])*Math.abs(subVector[2
])*Math.abs(subVector[1]));
  }
```

```
this.rigidBody.moveTo(jigLib.Vector3DUtil.create(this.position[0],t
his.position[1],this.position[2]));// Move the object to the
location of the object.
  var matrix=mat4.create();
  mat4.fromQuat(matrix,this.quaternion);
  var orient=new jigLib.Matrix3D(matrix);
  this.rigidBody.setOrientation(orient);//Change orientation of the
rigid body
  this.rigidBody.set_movable(false);
  this.system.addBody(this.rigidBody);
}
```

The preceding function, `initializePhysics`, takes a Boolean parameter sphere. If we pass sphere as true, then a `JSphere` collider is initialized; otherwise, a `JBox` collider is initialized. We set the mass of the object equivalent to its volume (radius for `JSphere` and *width depth height* for `JBox`). We then move the rigid body to the location of the `StageObject` class and also orient it in the `StageObject` class's direction. Note that we have declared all objects as static bodies (`this.rigidBody.set_movable(false);`).

The last change is added to our `updateMatrix` function. The `updateMatrix` function now needs to set the orientation and position of the `StageObject` class to the position and orientation of the `rigidBody` object. The following code only considers the `position` and `orientation` values of the `rigidBody` object when the rigid body is dynamic; otherwise, it sets the values from the `position` and `quaternion` parameters of the class. Our `updateMatrix` function is as follows:

```
StageObject.prototype.updateMatrix=function () {
  if(this.rigidBody&&this.rigidBody.get_movable()){
    var pos=this.rigidBody.get_currentState().position;
    this.position=vec3.fromValues(pos[0],pos[1],pos[2]);

mat4.copy(this.modelMatrix,this.rigidBody.get_currentState()._orien
tation.glmatrix);
  }else{
    mat4.identity(this.modelMatrix);
    mat4.fromQuat(this.modelMatrix,this.quaternion);
  }
  mat4.scale(this.modelMatrix,this.modelMatrix,this.scale);
  this.modelMatrix[12]=this.position[0];
  this.modelMatrix[13]=this.position[1];
  this.modelMatrix[14]=this.position[2];
  this.matrixWorldNeedsUpdate = true;
}
```

```
}
```

Now, we need to initialize our `rigidBody` object. We initialize it when the object is added to the `Stage` class. Open the `Stage.js` file from the `primitive` folder and go to the `addModel` function:

```
addModel:function(stageObject){
  if(!(this.gl===undefined))  {
    stageObject.createBuffers(this.gl);
  }
  var len= this.stageObjects.length;
  this.stageObjects.push(stageObject);
  if(stageObject.rigidBody){
    stageObject.rigidBody._id=len;
  }else{
    stageObject.system=this.system;
    stageObject.initializePhysics(false);
    stageObject.rigidBody._id=len;
    this.system.addBody(stageObject.rigidBody);
  }
},
```

The `addModel` function first checks whether `rigidBody` is already created. If it is already created, then we simply assign it an `_id` parameter. The `_id` parameter is the index of the `stageObject` in the `stageObjects` array. If `rigidBody` is not already created, it invokes the `initializePhysics` function of the `StageObject` class and then assigns it an ID. It then adds the object to the physics system.

Calculating the screen coordinates of a click

Getting the click coordinates on the canvas requires a little work. The property of the mouse click events, `offsetX` and `offsetY`, gives us the coordinates relative to the parent container. However, they are not very useful as they are not available in some browsers such as Firefox.

We will use `e.clientX`/`e.clientY`. They are event attributes that give us the horizontal and vertical coordinates (according to the current window) of the mouse pointer. Hence, we need to first calculate the position of our canvas with respect to the document, add the page scroll, and then subtract that value from the `e.clientX` attribute. Open the `09-Picking-Using-Ray-Casting.html` file in your editor. The following code snippet is present in this file:

```
var parent=document.getElementById('canvas');
parent.onmousedown=function(e) {
  var x, y, top = 0, left = 0;
  while (parent && parent.tagName !== 'BODY')
  {
    top =top+ parent.offsetTop;
    left =left+ parent.offsetLeft;
    parent= parent.offsetParent;
  }
  left += window.pageXOffset;
  top -= window.pageYOffset;
  x = e.clientX – left;
  y=(e.clientY – top);
  y = gl.viewportHeight - y;
  var rigidBody=rayCaster.pickObject(x,y,system);
...
};
```

In the preceding code, first we get the object of the canvas element and store it in the `parent` variable. The `offsetTop` and `offsetLeft` variables return the top and left values with respect to their parent. We then add the values to the `top` and `left` variables, and then we retrieve their parent using the `offsetParent` variable. We perform the preceding process until the `parent` variable becomes null or the `parent` variable is not the `BODY` tag. Hence, the `top` and `left` variables contain the sum of the `offsetTop` and `offsetLeft` variables of the canvas' parents. Then, we counter for the page scroll height (`left += window.pageXOffset` and `top -= window.pageYOffset`).

We subtract the canvas' location (`x = e.clientX – left;`) with respect to the client's view from the mouse click coordinate.

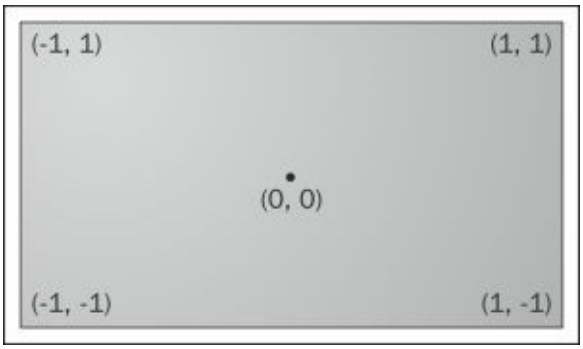
Note

We calculate the *y* value after subtracting it from the height of the canvas (`gl.viewportHeight`). We do so because the WebGL coordinate system originates in the bottom left, while the coordinate we calculated previously was taken from the top right of the screen.

We then pass the computed click coordinates to our `rayCaster.pickedObject` function, where we convert these values to **Normalized Device Coordinates** (**NDC**). We calculate the click coordinates as follows:

```
var x1 = ( x / this.screen_width )*2  - 1;
var y1 = ( y / this.screen_height)*2  - 1;
```

The following diagram shows how we represent our 3D scene on a 2D canvas. It shows that the origin of the world (0,0,0) is marked as the center (0,0) of the screen:



The NDC is a value that remains the same and is independent of the view size. It ranges from -1 to 1. Basically, the objective is that the coordinate that we want to convert to the world coordinate has to first be independent of the canvas size. Hence, we divide it by the height and width of the canvas. The conversion is shown in the following diagram:

Unproject the vector

Our NDC vector's coordinates look something like this (*x1*, *y1*, 1). This means that it is one unit ahead of the eye vector (*x1*, *y1*, 0). Basically in this case, the eye vector is the click coordinate. Now, this NDC has to be converted to world coordinates.

Throughout the book, we have discussed this formula:

$v' = P M v$

Each vertex is rendered after multiplying it with the ModelView matrix and then with the projection matrix of the camera. The final values vector (*v'*) has three components (*x*, *y*, *z*), where *x* and *y* denote the location on the screen and *z* is for depth test. Now, we have a device coordinate, *v'*, and we want to convert it to the world coordinate. So our formula should be as follows:

$v = M^{-1} * P^{-1} * v'$

This means that if we multiply the NDC with the inverse projection matrix, then with the inverse ModelView matrix of the camera, we convert the NDC to the world coordinate.

Our projection matrix and ModelView matrix are 4 x 4 matrices and our vector is a 1 x 3 vector. To multiply these matrices with our vector, we have to convert it to a 1 x 4 vector. Hence, we add a *w* component, 1. Now, our `vec3` vector becomes vec4 (*x1*, *y1*, 1, 1). After multiplication with the matrices, we get the *w* component which might not be equal to 1. We need to perform a **perspective divide** to normalize our coordinates. Hence, perspective divide is division of all four coordinates (*x*, *y*, *z*, *w*) by *w* in order to normalize the *w* value to 1.

So to unproject the vector and convert it from screen coordinates to world coordinates, we need to perform the following steps:

1. Multiply the vector by the inverse camera projection matrix.
2. Multiply the vector by the inverse camera ModelView matrix.
3. Perform perspective divide to normalize the *w* component.

To perform the preceding operations, we have added a function,

unProjectVector, to our camera class and a general purpose applyProjection function in utils.js. Open the utils.js file in your favorite editor. The following code snippet is present in this file:

```
function applyProjection(vector,m){

  var x = vector[0], y = vector[1], z = vector[2];
  var e = m;
  var d = 1 / ( e[3]  x + e[7]  y + e[11]  z + e[15] ); //
perspective divide
  var x1 = ( e[0]  x + e[4]  y + e[8]   z + e[12] )  d;
  var y1 = ( e[1]  x + e[5]  y + e[9]   z + e[13] )  d;
  var z1 = ( e[2]  x + e[6]  y + e[10]  z + e[14] ) * d;
  return vec3.fromValues(x1,y1,z1);
}
```

The preceding function, applyProjection, takes a vector (vec3) and matrix as parameters. It first calculates the *w* component (`var d = 1 / ( e[3] x + e[7] y + e[11] * z + e[15] );`), then multiplies each component, x1, y1, and z1, with it to get the normalized vector.

Open the camera.js file from the primitive folder in your editor to understand our new function, unProjectVector:

```
Camera.prototype.unProjectVector=function(vector){
  var inverseProjection=mat4.create();
  mat4.invert(inverseProjection,this.projMatrix);
  var viewProjection=mat4.create();
  mat4.invert(viewProjection,this.viewMatrix);
  mat4.mul(viewProjection,viewProjection,inverseProjection)
  var vec=applyProjection(vector,viewProjection);
  return vec;
}
```

In the preceding code, we first calculate our inverse projection matrix inverseProjection. We then calculate our inverse ModelView matrix and multiply them to get our viewProjection matrix.

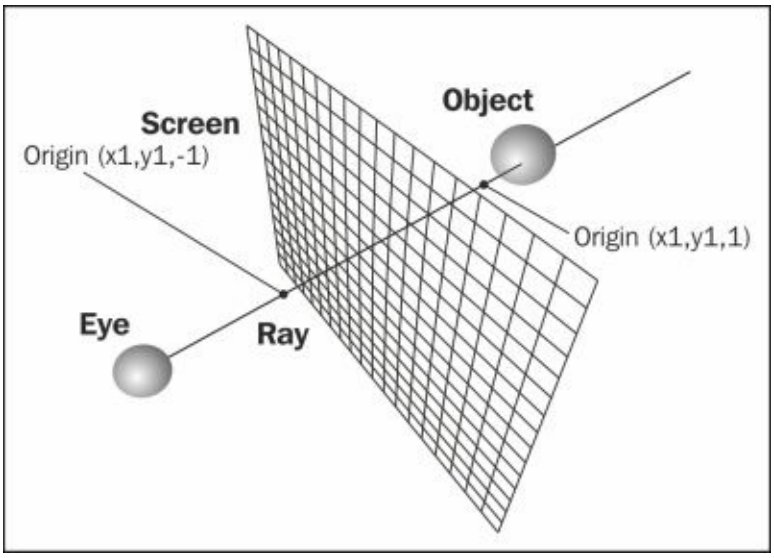
viewProjection = M-1 * P-1

Now we multiply our viewProjection matrix with the vector using our function applyProjection, which also performs the perspective divide.

Creating a ray segment

To create a ray segment, we need two components, origin and direction. The origin is straightforward. The NDC (*x1*, *y1*, 1) converted world coordinate will become our origin, but the direction is tricky. We either use the direction of the camera or alternatively, we can create two vectors with a varying *z* to calculate the direction.

In the following diagram, we have depicted two NDCs, `originVector` (`x1`, `y1`, `-1`) and `startVector` (`x1`, `y1`, `1`), lying on the light ray; one lying behind and the other ahead of the camera (eye vector). We subtract one vertex from another to get the direction:



Open the `RayCaster.js` file from `primitive/game` in your favorite editor. The following code snippet is present in this file:

```
RayCaster.prototype.makeRay=function(x,y){
  var x1 = ( x / this.screen_width )*2  - 1;
  var y1 = ( y / this.screen_height)*2  - 1;
  var startVector=vec3.fromValues(x1,y1,1);
  var originVector=vec3.fromValues(x1,y1,-1);
  var directionVector=this.camera.unProjectVector(startVector);
  var origin=this.camera.unProjectVector(originVector);
  vec3.sub(directionVector,directionVector,origin);
  vec3.normalize(directionVector,directionVector);
  //vec3.scale(directionVector,directionVector,-1);
```

```
  return {origin: origin, directionVector: directionVector};;
}
```

The function takes the canvas click coordinates (`x`, `y`) as parameters. It converts them to NDC (`x1`, `y1`) after dividing them with the height and width of the canvas. We create our two vectors: `startVector` (`x1`, `y1`, `1`) and `originVector` (`x1`, `y1`,`-1`). Then, we unproject both the vectors to convert them from NDC to world coordinates and store them in the `origin` and `directionVector` variables.

We calculate the direction vector after subtracting the origin from `directionVector` and storing the result in the direction vector (`vec3.sub (directionVector, directionVector, origin);`). As the direction vector is a unit vector, we normalize it (`vec3.normalize (directionVector, directionVector);`). We then return the `origin` and `directionvector` variables (`return {origin: origin, directionVector: directionVector};`) to the calling function.

Checking for an intersection

Once we have created our ray, all we have to do is invoke the `segmentIntersect` function of our collision system to get the intersecting object.

Open the `RayCaster.js` file from `primitive/game` in your favorite editor. The following code snippet is present in this file:

```
RayCaster=function(SCREEN_WIDTH,SCREEN_HEIGHT,camera){
  this.camera=camera;
  this.screen_height=SCREEN_HEIGHT;
  this.screen_width=SCREEN_WIDTH;
  this.pickedObject=null;
};
RayCaster.prototype.pickObject=function(x,y,system){
  var ray=this.makeRay(x,y);
  var directionVector=vec3.create();
  //The constant 100 is added to define the range of the ray. In
our implementation, the ray is not infinite.
  vec3.scale(directionVector,ray.directionVector,100)
  var segment=new jigLib.JSegment(ray.origin,directionVector);
  var out={};
  var cs=system.getCollisionSystem();
  if(cs.segmentIntersectGame(out, segment, null)){
    if(this.pickedObject){
      this.pickedObject.isPicked=false;
     }
    returnout.rigidBody;
  }
}
```

The constructor of the `RayCaster` class takes the width and height of the canvas and the camera object as parameters and stores them in the class variables.

The function, `pickObject`, takes the `x` and `y` canvas click coordinates and the physics system object (`system`). It first invokes the `makeray` function of its class to get the computed `origin` and `directionVector` variables and stores them in the ray object (`ray.origin`, `ray.directionVector`). Then, it scales `directionVector` by `100` as the `JSegment` class takes the `_origin` and `_delta` (length and direction of the vector) parameters.

Then, we create a segment object and invoke `cs.segmentIntersectGame(out, segment, null)`. If the function returns true, we unselect the active picked

object by setting its `isPicked` attribute to `false` and return the selected `rigidBody` object to the calling function. We have added another attribute, `isPicked`, to our `StageObject` class, so when a new object is picked, we can unselect the old one.

Changing the color of the selected object

Now we are at the climax of the story. We will now walk through the complete implementation to understand the changes in our main code to change the color of our selected/picked object.

Open `09-Picking-Using-Ray-Casting.html` in your editor.

We first create the object of our `RayCaster` class with the height and width of the canvas and the camera object as the attributes. The `start()` function is as follows:

```
function start() {
...
  rayCaster=new RayCaster(gl.viewportWidth,gl.viewportHeight,cam);
...
  var parent=document.getElementById('canvas');
  parent.onmousedown=function(e) {
    var x, y, top = 0, left = 0;
    while (parent&&parent.tagName !== 'BODY')
    {
      top += parent.offsetTop;
      left += parent.offsetLeft;
      parent    = parent.offsetParent;
    }
    left += window.pageXOffset;
    top -= window.pageYOffset;
    x = e.clientX - left;
    y = gl.viewportHeight - (e.clientY - top);

    var rigidBody=rayCaster.pickObject(x,y,system);

    stage.stageObjects[rigidBody._id].isPicked=true;
    rayCaster.pickedObject=stage.stageObjects[rigidBody._id];
  };.............
}
```

We attach an `onmousedown` event to our canvas object. When the mouse is clicked, we calculate the mouse click's *x* and *y* coordinates using the sum of `offsetLeft`/`offsetTop` of the canvas's parent objects. Then, we invoke the `pickObject` function of our `rayCaster` object and the `pickObject` function returns the object of the selected `rigidBody` object. The `_id` parameter of the `rigidBody` object is retrieved. This `_id` parameter is the index of the

`stageObject` in the `stageObjects` array. We retrieve the selected `stageObject` and set its `isPicked` property to `true` (`stage.stageObjects[rigidBody._id].isPicked=true;`) and store the selected `stageObject` in the `pickedObject` attribute of the `RayCaster` class (`rayCaster.pickedObject=stage.stageObjects[rigidBody._id];`). The `pickedObject` attribute of the `RayCaster` class is used to unset the `isPicked` property of the object when a new object is selected.

Remember all stage objects have an associated `rigidBody` object, which is initialized when the `addModel` function of the `Stage` class is invoked with the `stageObject` as the parameter.

Our `drawScene` function is as follows:

```
function drawScene() {
...
  for(var i=0;i<stage.stageObjects.length;++i){
...
    if(stage.stageObjects[i].isPicked)
    {
      gl.uniform3f(shaderProgram.materialDiffuseColor,0,255,0);

    }
    else
    {

gl.uniform3f(shaderProgram.materialDiffuseColor,stage.stageObjects[
i].diffuseColor[0],stage.stageObjects[i].diffuseColor[1],stage.stag
eObjects[i].diffuseColor[2]);
    }

if(stage.stageObjects[i].materialFile!=null&&!stage.stageObjects[i]
.isPicked){
    gl.uniform1i(shaderProgram.hasTexture,1);
....
    }
    else{
      gl.uniform1i(shaderProgram.hasTexture,0);

gl.disableVertexAttribArray(shaderProgram.textureCoordAttribute);
    }
...
  }
}
```

We have also modified our `drawScene` function to change the color of our picked object. While iterating over our array of `stageObjects`, if any object's `isPicked` attribute is `true`, we set our own diffuse color (`gl.uniform3f(shaderProgram.materialDiffuseColor, 0, 255, 0);`). We do not use the diffuse color loaded from the JSON file. We apply a texture only if the `isPicked` property is `false`. Earlier, we only checked if the `materialFile` variable was associated with it, but now we also check for the `isPicked` property.

Note

The `jigLib.JSegment` class was created only for ray casting and not picking. In many physics engines, you will see that in the picking code, the ray takes only the direction and not the length of the ray (infinite ray). Hence, when you test the picking, you will have to steer the camera close to the object using arrow keys to pick it.

Offscreen rendering using framebuffers

WebGL games sometimes require the rendering of images without displaying them to the user or screen. For example, a game may want to draw the offline image and then cache it to save the reprocessing time. Offscreen rendering is also used to generate shadows of objects. Another example; our game might want to render an image and use it as a texture in a filter pass. For best performance, offscreen memories are managed by WebGL. When WebGL manages offscreen buffers, it allows us to avoid copying the pixel data back to our application.

WebGL has framebuffer objects. The framebuffers allow our game to capture a game scene in the GPU memory for further processing.

A **framebuffer object** (**FBO**) is a drawable object. It is a window-agnostic object that is defined in the WebGL standard. After we draw to a framebuffer object, we can read the pixel data to our game or use it as source data for other WebGL commands.

Framebuffer objects have a number of benefits. They are easy to set up, save memory, and are associated with a single WebGL context. We can use them for 2D pixel images and textures. The functions used to set up textures and images are different. The WebGL API for images uses renderbuffer objects. A renderbuffer image is simply a 2D pixel image. To explain this better, a texture has only the associated color value but a 2D pixel image has a color value and depth information.

We store the color data in a WebGL texture object but store the depth value in a renderbuffer object.

Creating a texture object to store color information

In the following code, first we create a texture object. The texture object is bound to context for further processing. See Chapter 4, *Applying Textures*, to understand the parameters. Also, notice that the last parameter, `null`, is an API call `gl.texImage2D`. In this parameter, we generally pass the image data, but as we are creating the texture buffer for storing data, we pass `null` to just create memory. Also, the width and height are that of the canvas:

```
frametexture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, frametexture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, width, height, 0, gl.RGBA,
gl.UNSIGNED_BYTE, null);
```

Creating a renderbuffer for depth information

We create a renderbuffer and bind it to apply further functions on it. Then, we allocate the storage for the renderbuffer. It is of the same size as the texture, with 16-bit storage for the depth component. Other options are `gl.DEPTH_COMPONENT8` and `gl.DEPTH_COMPONENT24`. The `createRenderbuffer()` function is as follows:

```
renderbuffer = gl.createRenderbuffer();
gl.bindRenderbuffer(gl.RENDERBUFFER,renderbuffer);
gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
width, height);
```

Associating a texture and a renderbuffer to framebuffers

We first create our framebuffer. We associate our texture using the `framebufferTexture2D` API call and then associate our renderbuffer with the `framebufferRenderbuffer` API call:

```
framebuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
gl.TEXTURE_2D, frametexture, 0);
gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
gl.RENDERBUFFER, renderbuffer);
```

Then, we unbound the framebuffer and the renderbuffer as follows:

```
gl.bindTexture(gl.TEXTURE_2D, null);
gl.bindRenderbuffer(gl.RENDERBUFFER, null);
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
```

Rendering to framebuffers

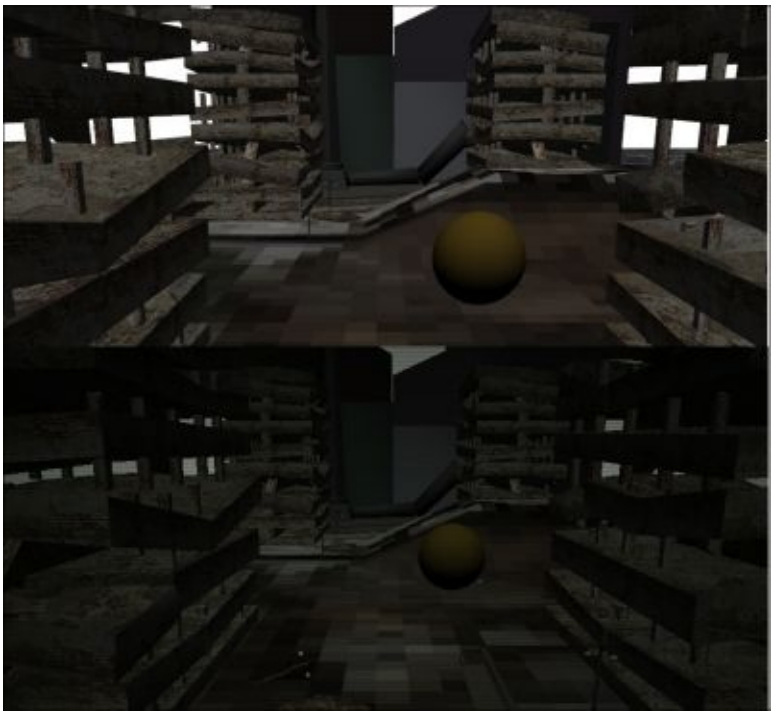
If we want to render to the framebuffer, we simply make it the bound buffer object. Remember the user display also happens in the display framebuffer, which is the active bound buffer or default buffer. If we make any other buffer the active bound buffer, it becomes the rendering target and when we want to render to the screen, we simply unbind it and the default buffer becomes the active bound buffer. The `bindFramebuffer` function is as follows:

```
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
drawScene();
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
```

Applying filters using framebuffers

Framebuffers are a very powerful extension to the WebGL API. A whole separate book can be written to explore its power, but we would like to confine ourselves to exploring it in a use case that is mostly used in game engines: post processing filters. So, let's understand what they are. Let's say that you want to render your game scene and give a user the perception that your game is being viewed using night vision goggles. Now, the night vision goggles give a perception of the green color all over the rendered scene. Let's say you want to give the perception of day and night in your game without creating static texture assets. There are many filter effects such as blur, glow, snow, and edge detection that we can apply.

In our example, to keep things simple, we add scan lines to our scene. We would like our scene to appear similar to the following screenshot:



If you notice in the preceding screenshot, we have rendered the scene without applying filters and another one with filters. The scene in the lower section of the preceding screenshot has a shade of green with alternate scan lines with dark

and light green colors. We used a very simple algorithm to produce this effect so that our focus is on how we do it in WebGL and not in the shader algorithm. You can use many complex algorithms in shaders to produce this effect such as edge detection using the Sobel operator (http://en.wikipedia.org/wiki/Sobel_operator).

So, let's understand the algorithm of filters:

1. Create a vertex and fragment shader to process the image.
2. Create a framebuffer.
3. Render the scene in the framebuffer with the associated texture and renderbuffer.
4. Use the new shaders to render the image.
5. Render the processed image.

In the preceding steps, we never render the scene. In fact, we render only the processed image. Some might think that our click events and other things would not work because we are rendering the image, which is not true. Basically, we process or create the framebuffer on every frame and then display the processed texture.

Also note that the first step tells you to create new shaders. The new shaders are used only to render the image in a square primitive. Now, we will have two different shaders, one set of fragment and vertex shaders to render the scene in the framebuffer and a new shader to render the image.

We will add two very simple shaders in our code and also change our main control code to load and link these shaders.

The vertex shader

The following vertex shader is pretty straightforward. It takes texture coordinates and the model vertices as attributes. We store the texture coordinate in a varying, `vTextureCoord`, to be processed by the fragment shader:

```
<script id="scanlines-vertex-shader" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec2 aTextureCoord;
varying vec2 vTextureCoord;
void main(void) {
  vTextureCoord = aTextureCoord;
  gl_Position = vec4(aVertexPosition,1.0);
}
</script>
```

The fragment shader

The fragment shader uses a simple algorithm to add scan lines. First, we read the color of the texture at the `vTextureCoord` location. Then, based on the `y` value of the texture coordinate, we decide if we want to multiply our color value with a factor (`0.80`) or not. We play with the *y* coordinate as we want horizontal scan lines. If we wanted vertical scan lines, we would have manipulated the color based on the *x* coordinate. The following code is for the fragment shader:

```
<script id="scanlines-fragment-shader" type="x-shader/x-fragment">
precisionhighp float;
uniform sampler2D uSampler;
const float canvasHeight=650.0;
varying vec2 vTextureCoord;
void main(void)
{
  vec4 color = texture2D(uSampler, vTextureCoord);
  if(mod(vTextureCoord.y,(4.0/canvasHeight))>(1.0/canvasHeight))
  color.rgb*=0.80;
  color.rgb*=pow(1.0-length(vTextureCoord.xy-0.5),1.3);
  color.rgb*=vec3(0.93,1.0,0.93);
  gl_FragColor = vec4(color.rgb,1.0);
}
</script>
```

The fragment shader is where the post-processing happens. Let's understand how it works. In the following fragment shader, we are not manipulating the texture color. This would render the scene exactly the same. So basically, we pass the texture from the framebuffer as a uniform to the shader and the shader simply displays the texture as shown in the following code:

```
<script id="scanlines-fragment-shader" type="x-shader/x-fragment">
precisionhighp float;
uniform sampler2D uSampler;
varying vec2 vTextureCoord;
void main(void)
{
  vec4 color = texture2D(uSampler, vTextureCoord);
  gl_FragColor = vec4(color.rgb,1.0);
}
</script>
```

Now, in the following fragment shader, we simply reverse the color of the textures. We subtract each texture color component from `1` as shown in the

following code:

```
<script id="scanlines-fragment-shader" type="x-shader/x-fragment">
precisionhighp float;
uniform sampler2D uSampler;
varying vec2 vTextureCoord;
void main(void)
{
  vec4 color = texture2D(uSampler, vTextureCoord);
  gl_FragColor = vec4(1-color.r,1-color.g,1-color.b,1.0);
}
</script>
```

So basically, the complete post-processing of the image happens in the fragment shader, although we can also preprocess an image in the main code by using the `readPixels` function of the WebGL API. However, this would add to the processing time as we would be iterating over the pixels of the image twice: once in the control code and the second time in the fragment shader. However, in some cases, it might be unavoidable.

Loading and linking shaders

We have modified our code to load and link two shaders. We added a generic function which loads common attributes and uniforms, required for both shaders. The common attributes are `aVertexPosition` for vertex positions and `aTextureCoord` for texture coordinates. The common uniform is `uSampler` for the texture. The `initShaderCommon` function is as follows:

```
function initShaderCommon(vertexShaderID,fragmentShaderID,prg){
  var fragmentShader = getShader(gl,vertexShaderID );//"shader-fs"
  var vertexShader = getShader(gl, fragmentShaderID);//"shader-vs"
  gl.attachShader(prg, vertexShader);
  gl.attachShader(prg, fragmentShader);
  gl.linkProgram(prg);
  if (!gl.getProgramParameter(prg, gl.LINK_STATUS)) {
    alert("Could not initialiseshaders");
  }
  prg.vertexPositionAttribute = gl.getAttribLocation(prg,
"aVertexPosition");
  prg.textureCoordAttribute = gl.getAttribLocation(prg,
"aTextureCoord");
}
```

The preceding function, `initShaderCommon`, takes three parameters: `vertexShaderID`, `fragmentShaderID`, and `prg`. The `vertexShaderID` and `fragmentShaderID` parameters are element IDs of the script tag, which are described in the following code:

```
<script id="scanlines-vertex-shader" type="x-shader/x-vertex">
...
</script>
<script id="scanlines-fragment-shader" type="x-shader/x-fragment">
...
</script>
```

Our two new shader programs are defined by IDs, `"scanlines-fragment-shader"` and `"scanlines-vertex-shader"`. The `prg` object is the shader program object to hold the compiled program:

```
var shaderProgram;
var postProcessShaderProgram;
```

Hence, we have created two shader program objects for two different shader programs:

```
postProcessShaderProgram = gl.createProgram();
initShaderCommon("scanlines-vertex-shader","scanlines-fragment-
shader",postProcessShaderProgram);
...
shaderProgram = gl.createProgram();
initShaderCommon("shader-vs","shader-vs",shaderProgram)
...
}
```

We have two shader programs. We initialize them, link them, and attach attributes to them. Note that we do not use either of them for rendering. We only use them for rendering when we say `gl.useProgram(postProcessShaderProgram)`. What we are trying to emphasize is a very useful concept in powerful game engines. A good game engine has many shaders. They compile some shaders and link them, but use those shaders as and when required.

In our case, we are using one shader to render the complete scene to the framebuffer (`shaderProgram`) and the other shader to render the texture (`postProcessShaderProgram`). Big games use different shaders to render different objects in the scene. In our render loop, when we iterate over the list of `stageObjects`, we can use different shaders for each object by invoking the `gl.useProgram` function.

Understanding the square geometry code

We have added a new geometry, a square. This geometry will be used to render our texture. Open `SquareGeometry.js` from the `primitive` folder in your editor.

The `SquareGeometry` class defines four corner vertices and four UV coordinates. Then we declare our two faces of two triangles. The indices of the two triangles for each face are [0,1,2] and [2,1,3]:

```
SquareGeometry = inherit(Geometry, function ()
{
  superc(this);
  this.vertices=[
    -1.0,-1.0,0.0,//0
    1.0,-1.0,0.0,//1
    -1.0, 1.0,0.0,//2
    1.0, 1.0,0.0//3
  ];
  var uvs=[ 0.0, 0.0,
    1.0, 0.0,
    0.0, 1.0,1.0,1.0];
  var face = new Face();
  face.a=0;
  face.b=1;
  face.c=2;
  this.faces.push( face );
  var face = new Face();
  face.a=2;
  face.b=1;
  face.c=3;
  this.uvs[0]=uvs;
  this.indicesFromFaces();
});
```

The other class we have added is `Square` which inherits the `StageObject` class. Open `Square.js` from the `primitive` folder in your editor. We just added this class to keep the architecture intact and also to reuse the buffer creation code:

```
Square= inherit(StageObject, function (){
  superc(this);
  this.geometry=new SquareGeometry();
  this.materialFile="framebuffer";
});
```

Implementing the filter

Let's put all the pieces together. Open `09-Filters-FrameBuffer-Scan-Lines.html` in your editor.

We first declare all our variables as follows:

```
var shaderProgram;
var postProcessShaderProgram;
...
var frametexture = null;
var framebuffer = null;
var renderbuffer = null;
var square=null;
```

The first two variables are used to hold shader program objects while the next three variables are used to hold the texture, framebuffer, and renderbuffer objects.

The `bindPostProcessShaderAttributes()` function is as follows:

```
function bindPostProcessShaderAttributes(){
  postProcessShaderProgram = gl.createProgram();
  initShaderCommon("scanlines-vertex-shader","scanlines-fragment-
shader",postProcessShaderProgram);
  square=new Square();
  square.createBuffers(gl);
  createFrameBuffer();
}
```

The preceding function initializes the shader program and initializes the `SquareGeometry` class. We also invoke our `createFrameBuffer()` function. The `createFrameBuffer()` function does exactly as expressed in section *Offscreen rendering using framebuffers*. The only important section of the code has been listed here. The `gl.texImage2D` function is invoked with the `null` parameter because memory is being allocated. We create a renderbuffer and also create memory to hold texture depth using the API call `renderbufferStorage`. We then associate the renderbuffer and texture to our framebuffer object.

The important thing to note is that the `frametexture` and `renderbuffer` variables take the size (`height`, `width`) of the viewport:

```
functioncreateFrameBuffer(){
```

```
  var width=gl.viewportWidth;
  var height=gl.viewportHeight;
  frametexture = gl.createTexture();
//...
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, width, height, 0,
gl.RGBA, gl.UNSIGNED_BYTE, null);
  renderbuffer = gl.createRenderbuffer();
//...
  gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
width, height);
  framebuffer = gl.createFramebuffer();
  gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
  gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
gl.TEXTURE_2D, frametexture, 0);
  gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
gl.RENDERBUFFER, renderbuffer);
...
}
```

The `start()` function initializes the shaders and invokes the `bindPostProcessShaderAttributes()` function:

```
function start() {
...
  initShaders("shader-vs","shader-fs");
  bindPostProcessShaderAttributes();
...
}
```

The last part is the rendering logic. We changed our animate function to invoke the `draw()` function instead of the `drawScene()` function:

```
function animate(){
...
  draw();
...
}
```

The `draw()` function now does two things:

- Makes our framebuffer the active bound buffer so that the scene can render offscreen and then invokes `drawScene()` to render the scene.
- Unbinds the framebuffer to use the default display framebuffer and then invokes `drawScenePostProcess()`. Now the texture is drawn on the screen.

The `draw()` function is defined as follows:

```
function draw(){
  // resizeSize();
  gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
  drawScene();
  gl.bindFramebuffer(gl.FRAMEBUFFER, null);
  //Postprocessing
  drawScenePostProcess();
}
```

The `drawScene()` function is modified to use the `shaderProgram` parameter to render the scene:

```
functiondrawScene() {
  gl.useProgram(shaderProgram);
...
}
```

The `drawScenePostProcess()` function scene basically prepares attributes and uniforms to render the texture on the square. It also switches to the new shader using `gl.useProgram`:

```
function drawScenePostProcess() {
  gl.useProgram(postProcessShaderProgram);

gl.enableVertexAttribArray(postProcessShaderProgram.vertexPositionA
ttribute);
  gl.bindBuffer(gl.ARRAY_BUFFER, square.vbo);

gl.vertexAttribPointer(postProcessShaderProgram.vertexPositionAttri
bute, square.vbo.itemSize, gl.FLOAT, false, 0, 0);

gl.enableVertexAttribArray(postProcessShaderProgram.textureCoordAtt
ribute);
  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, frametexture);
  gl.uniform1i(gl.getUniformLocation(postProcessShaderProgram,
"uSampler"), 0);
  gl.bindBuffer(gl.ARRAY_BUFFER, square.verticesTextureBuffer);

gl.vertexAttribPointer(postProcessShaderProgram.textureCoordAttribu
te, 2, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, square.ibo);
  gl.drawElements(gl.TRIANGLES, square.geometry.indices.length,
gl.UNSIGNED_SHORT,0);
}
```

The preceding function informs WebGL to use the new shader program. It then activates the vertex buffer object and index buffer object of the `square` object and associates the indices and vertices to the buffers to be processed by the shader. It also uploads the `frameTexture` object to the zeroth texture location. The zeroth location is associated with the `uSampler` uniform to be processed in the shader. Note that the `frameTexture` object is associated with the `framebuffer` object which in turn is populated in the `drawScene()` call.

Summary

In this chapter, we covered four very frequently used concepts in game development. The first part of the chapter covered two concepts, picking and ray casting. Ray casting is the most widely used concept for game AI and collision detection. Picking is mostly used by CAD software but is a very powerful tool to build user preference screens in games. For instance, you use picking to let users pick car parts in a car racing game where you can actually display parts in 3D.

The next two concepts we covered were the use of framebuffers to apply filters in games and the use of multiple shaders in a game. Both concepts are very powerful to help you build high performance games. In games, we generally have a generic shader to render objects. In some cases, we might need specific shaders to render an object such as a shader to render the main character or add particle effects in a game. Like in our case, we have added one more shader for filters/post processing.

In the next chapter, we will discuss the 2D canvas and how it is helpful to create sprite labels in a game. We will also touch upon multiplayer games.

Chapter 10. 2D Canvas and Multiplayer Games

In all our previous chapters, we have discussed 3D rendering in-depth but have not discussed 2D rendering. In this chapter, we will discuss the canvas 2D context, which is a very important aspect of game development. The canvas 2D context offers a very powerful drawing API.

We will also discuss the concepts of HTML-based multiplayer games. We will then discuss Node.js and the Socket.IO API to implement a spectator player who can see what is happening in the game. In this chapter, we will cover:

- Canvas 2D basics and the drawing API
- 2D sprites for model labels and game scores
- Real-time communication: HTTP long polling and WebSockets
- Node.js: Socket.IO
- Sample implementation of the multiplayer game: a spectator player

Understanding canvas 2D basics and the drawing API

The 2D context provides objects, methods, and properties to draw and manipulate graphics on the drawing surface of a canvas. Refer to the link http://www.w3schools.com/html/html5_canvas.asp to learn more about canvas 2D. The following is some example code for the `canvas` element:

```
<html>
<canvas id="canvasElement"></canvas>
</html>
<script>
this.canvas = document.getElementById("canvasElement");
this.ctx = this.canvas.getContext('2d');
</script>
```

The `canvas` element has no drawing properties in itself and we use a script to draw graphics. The `getContext` method returns the object, which provides methods to draw text, lines, shapes, and graphics on the canvas.

The following code walks you through the methods of the 2D context object returned from the `getContext` method:

```
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.fillStyle="#FF0000";
ctx.fillRect(0,0,10,10);
```

The `fillstyle` property will fill the rectangle with the red color. The following code gets an image object and fills `rect (0,0,100,100)` with the image:

```
var img=document.getElementById("ball");
var patttern=ctx.createPattern(img,"repeat");
ctx.rect(0,0,100,100);
ctx.fillStyle=patttern;//fill pattern image
ctx.fill();
```

The following code sets fill color and text alignment and then draws the text at the given coordinates, `textX` and `textY`. The `ctx.measureText(txt).width` function is a text metrics function. It will give you the width of the text at the specified font style and font size:

```
ctx.fillStyle = #FF0000; // This determines the text color, it can
take a hex value or rgba value (e.g. rgba(255,0,0,0.5))
ctx.textAlign = "center";       // This determines the alignment of
text, e.g. left, center, right
ctx.textBaseline = "middle";
ctx.font="14px Georgia";
var txt="My width";
ctx.fillText(txt+ctx.measureText(txt).width, textX,  textY);
```

The following code introduces another powerful concept of transforms in the drawing API. The `ctx.transform()`function sets the transformation matrix before the next drawing API call:

```
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.fillStyle="green";
ctx.fillRect(0,0,250,100)
ctx.save();//Saves the state
ctx.transform(1,0.5,-0.5,1,30,10);ctx.fillStyle="black";
ctx.fillRect(0,0,250,100);
ctx.restore();// restores the state
```

The transformation matrix of a 2D object is denoted by a 3 x 3 matrix, which is shown as follows:

Scale Horizontally (a)	Skew Vertically (c)	Moves Horizontally (e)
Skew Horizontally (b)	Scale Vertically (d)	Moves Vertically (f)
0	0	1

The preceding matrix describes the parameters of the `transform()` function call. Also, the canvas 2D object is a state machine. The next `transform()` function call will act upon the previous transformation. Hence, we use the `ctx.save()` function to store the current state and make our transform call and then invoke the `ctx.restore()` function to revert to the previous state. The store function call does not just store the transformations but also other state values such as the fill color.

For further understanding of the 2D context, refer to http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#2dcontext.

Using canvas 2D for textures

Canvas 2D can be used for many 2D HTML games. If a canvas 2D object is not a part of the WebGL specification, then why are we discussing it?

This is because a canvas 2D object can be directly passed as a texture to the WebGL texture API call. In the following code, if you notice in the `gl.texImage2D` WebGL API call, we pass the canvas object as the image data parameter (`document.getElementById("canvasElement")`):

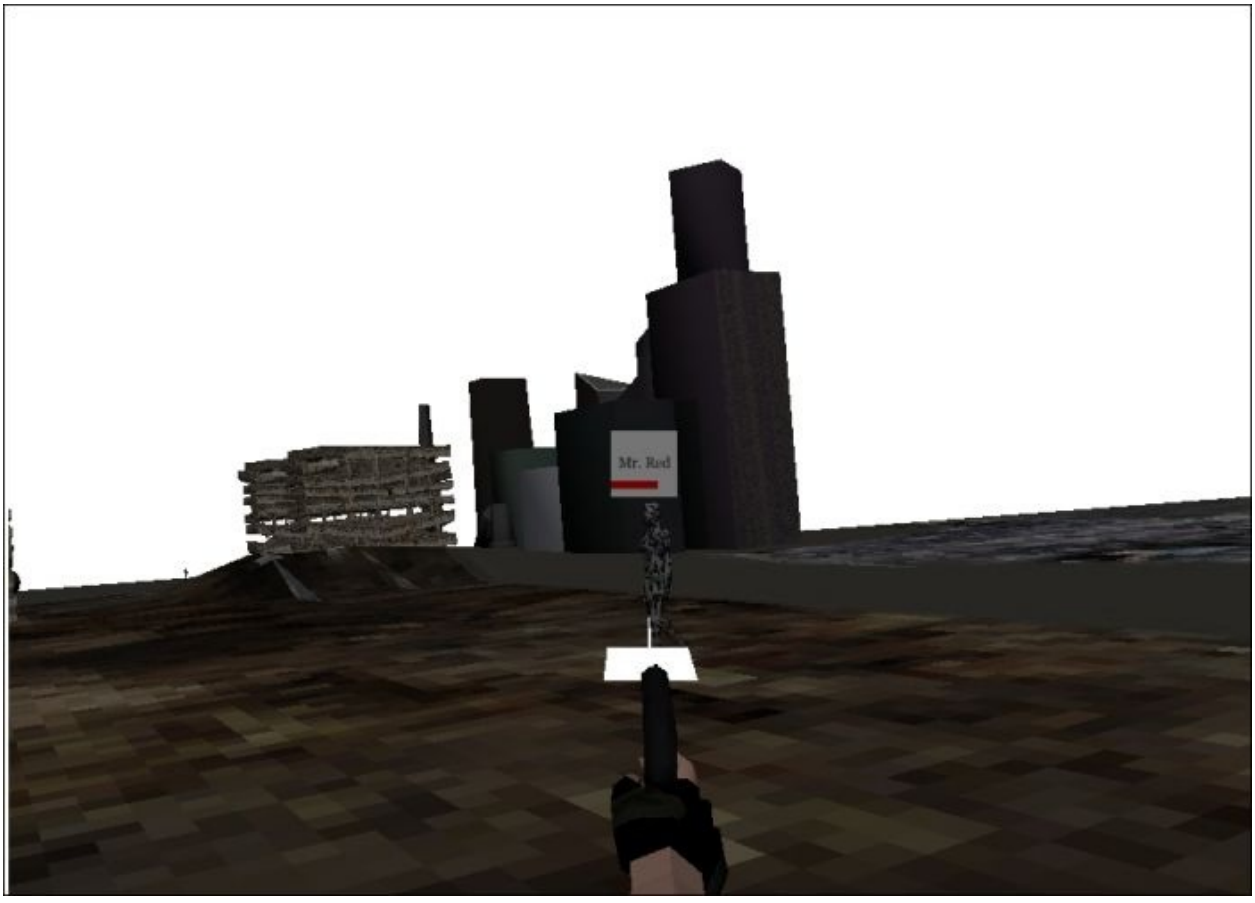
```
<html>
<canvas id="canvasElement"></canvas>
</html>
var texture=this.gl.createTexture();
this.gl.bindTexture(this.gl.TEXTURE_2D, texture);
this.gl.texImage2D(this.gl.TEXTURE_2D, 0, this.gl.RGBA,
this.gl.RGBA,
this.gl.UNSIGNED_BYTE,document.getElementById("canvasElement"));
this.gl.generateMipmap(this.gl.TEXTURE_2D);
this.gl.texParameteri(this.gl.TEXTURE_2D,
this.gl.TEXTURE_MAG_FILTER, this.gl.LINEAR);
this.gl.texParameteri(this.gl.TEXTURE_2D,
this.gl.TEXTURE_MIN_FILTER, this.gl.LINEAR_MIPMAP_NEAREST);
this.gl.bindTexture(this.gl.TEXTURE_2D, null);
```

Canvas 2D offers a very powerful drawing API, which can be used to generate dynamic textures in our game. It has numerous utilities such as manipulating images to drawing 2D text in our 3D game. WebGL does not directly offer us functions to draw text or manipulate pixel data in an image, hence the canvas 2D API comes in handy.

We will use the canvas 2D API to draw game stats and model labels in our code.

Adding 2D textures as model labels

In this section, we will discuss how we use dynamic textures to add labels to players in a game. In a multiplayer game environment, you may need to label your players with their usernames. Also, if you want to add stats overlay to your game, you can do it using two methods; either we use CSS overlays over our game canvas or use texture sprites and render it inside the canvas. If we plan to use the second method, we need a texture with text. Let's take a look at the following screenshot:



In the preceding screenshot, we have a label **Mr. Red** placed on top of our model and a red bar denoting the remaining power in the model (game stats). Now, the sprite that displays this information is a quad/square geometry with a dynamic texture. Also, the sprite faces the camera after it has been rotated. Hence, the sprite class should have the following features:

- It should be placed at a defined delta distance from the model and should move with the model.
- It should rotate when the camera rotates.

So to achieve the preceding features, we need three things: a quad geometry, a dynamic texture (canvas object), and a sprite class that inherits the `StageObject` class and binds the geometry to the texture.

Using the sprite texture

We will start our learning with evolving the sprite texture, which writes the text (model name) and generates the power bar.

We will try to build a generic class that can handle the following:

- The multiline text, if the text exceeds the maximum width
- Calculates the dynamic height and width of the sprite (canvas), based on font style, font size, and length of the text

Open the `SpriteTexture.js` file from `client/primitive` in the code files of this chapter in your text editor. The constructor takes the canvas element, `elemId`, as the parameter to render the text in:

```
SpriteTexture = function (elemId) {
  this.canvas = document.getElementById(elemId);
  this.ctx = this.canvas.getContext('2d');
  this.textSize=56;
  this.maxWidth=256;
  this.backgroundColor="#ffffff";
  this.color="#333333";
  this.powerColor="#FF0000";
  this.power=7;
  this.fullPower=10;
  this.squareTexture=false;
}
```

It first gets the object of the canvas element and then gets the reference to its context object. The default values of `backgroundColor`, `color` (text color), and `powerColor` are defined in the constructor. Also, the `maxWidth` value of the sprite is defined, in case the text is very long. The `this.power` variable holds the relative strength value with maximum being `10` (`this.fullPower`).

Then, we define two functions, `SpriteTexture.prototype.createMultilineText` and `SpriteTexture.prototype.measureText`. The `measureText` function is shown in the following code snippet:

```
SpriteTexture.prototype.measureText=function(textToMeasure) {
  return this.ctx.measureText(textToMeasure).width;

}
```

The `measureText` function is straightforward. It uses the `ctx.measureText` function of the canvas 2D API to get the width of the text.

The `createMultilineText` function is shown in the following code snippet:

```
/*
text is an array that holds the split multiline text.
TextData holds the text data to be split.
*/
SpriteTexture.prototype.createMultilineText=function(textData,text)
{
  textData = textData.replace("\n"," ");
  var currentText = textData;
  var newText;
  var subWidth = 0;
  var maxLineWidth = 0;
  var wordArray = textData.split(" ");
  var wordsInCurrent, wordArrayLength;
  wordsInCurrent = wordArrayLength = wordArray.length;
  while(this.measureText(currentText)
>this.maxWidth&&wordsInCurrent> 1) {
    wordsInCurrent--;
    var linebreak = false;
    currentText = newText = "";
    //currentText holds the string that fits in max width
    //newText holds the string left.
    // The loop iterates over the words to copy the words to
current text or new text
    for(var i = 0; i<wordArrayLength; i++) {
      if (i<wordsInCurrent) {
        currentText += wordArray[i];
        if (i+1 <wordsInCurrent) { currentText += " "; }
      }
      else {
        newText += wordArray[i];
        if( i+1 <wordArrayLength) { newText += " "; }
      }
    }
  }
  text.push(currentText);
  maxLineWidth = this.measureText(currentText);
  if(newText) {
    subWidth = this.createMultilineText(newText,text);
    if (subWidth>maxLineWidth) {
      maxLineWidth = subWidth;
    }
  }
```

```
  return maxLineWidth;
}
```

The preceding function, `createMultilineText`, performs two tasks:

- Stores split lines of text in the `text` array
- Returns the maximum width of the string with the `text` array

The function first replaces the newline (`\n`) characters with spaces and then splits the text into words and stores them in the `wordArray` array.

The `while` loop iterates until the length of the `currentText` variable is not less than the `maxWidth` variable. Basically, the code in the loop first concatenates the `wordsInCurrent` (which is 1 subtracted from the value of `wordArrayLength`) words in the `currentText` variable and concatenates the rest of the words in the `newText` variable. It then checks the length of `currentText`, if it is still greater than the value of `maxWidth`. Then, it copies `wordsInCurrent` (which is 2 subtracted from the value of `wordArrayLength`) to see if it fits. It continues to copy one word less to `currentText` until the width of the text is less than the value of `maxWidth`. Then, it invokes itself recursively with the `newText` variable (concatenated words greater than the value of `maxWidth`), which holds the rest of the words after storing the `currentText` variable in the `text` array. It also compares the length of the returned `subWidth` variable with `maxLineWidth`, to get the width of the longest string in the `text` array.

We also added a `powerOfTwo` function to calculate the width of canvas:

```
SpriteTexture.prototype.powerOfTwo=function(textWidth, width) {
  var width = width || 1;
  while(width<textWidth) {
    width *= 2;
  }
  return width;
}
```

This function is important as most WebGL implementations require that the texture dimensions must be in the power of two. Now if the width of our text becomes 150, then our canvas size has to be 256. If the width is 100, then the width of the canvas has to be 128. The width of the canvas is equal to the size of texture. This function sets the width to `1` and multiplies it by `2` until the width is greater than the passed `textWidth` variable.

The complete drawing happens in the `createTexture` function, which is defined as follows:

```
SpriteTexture.prototype.createTexture=function(textData){
  var text=[];
  var width=0;
  var canvasX=0;
  var canvasY=0;
  //Sets the font size
  this.ctx.font=this.textSize +"px Georgia";
```

If the text width is greater than the value of `maxWidth`, then we invoke the `createMultilineText` function to store the split text in the `text` array; otherwise, we store the string in the `text` array. Basically, the `text` array stores the string lines to be rendered. For the multiline text, we use the width returned by the `createMultilineText` function, otherwise we use the width of the passed string using the canvas API's `measureText` function. We pass the calculated width to the `powerOfTwo` function to calculate the appropriate width. The following code snippet explains the working of the the `createTexture` function:

```
// If the text width is greater than maxWidth, then call multiline,
else, store the string in text array.
if (this.maxWidth&this.measureText(textData) >this.maxWidth ) {
  width = this.createMultilineText(textData,text);
  canvasX = this.powerOfTwo(width);
} else {
  text.push(textData);
  canvasX = this.powerOfTwo(this.ctx.measureText(textData).width);
}
```

We calculate `canvasY` with the font size (`this.textSize`), multiplied by the length of the `text` array plus one. We also add another 30 units, to add more length of the power bar:

```
canvasY = this.powerOfTwo(this.textSize*(text.length+1)+30);//+30
for power stats
```

Sometimes, we may require a square texture. In that case, we set the height or width to the greater of the two:

```
if(this.sqaureTexture){
  (canvasX>canvasY) ? canvasY = canvasX : canvasX = canvasY;//For
squaring the texture
}
this.canvas.width = canvasX;
```

```
this.canvas.height = canvasY;
```

We calculate the location of the text. Note that we set the *x* location of the text to the half of the width of the canvas. This is because we set the text alignment to center using the following code:

```
var textX = canvasX/2;
var textY = canvasY/2;
this.ctx.textAlign = "center";// Sets the alignment of text, e.g.
left, center, right
this.ctx.textBaseline = "middle";
```

We paint the background of the canvas using the `ctx.fillRect` API call:

```
this.ctx.fillStyle = this.backgroundColor;
this.ctx.fillRect(0, 0, this.ctx.canvas.width,
this.ctx.canvas.height);
```

We set the font color using the following code:

```
this.ctx.fillStyle = this.color; // Sets the text color, it can
take a hex value or rgba value (e.g. rgba(255,0,0,0.5))
```

We draw each line of the text by iterating over the `text` array and we calculate `textY` for each string line using the following code:

```
this.ctx.font=this.textSize +"px Georgia";
var offset = (canvasY - this.textSize*(text.length+1))  0.5;
for(var i = 0; i<text.length; i++) {
  if(text.length> 1) {
    textY = (i+1)this.textSize + offset;
  }
  this.ctx.fillText(text[i], textX,  textY);
}
```

Then, we draw our power bar based on the value of power using the following code:

```
this.ctx.fillStyle = this.powerColor;
this.ctx.fillRect(0, this.canvas.height-60,
this.power*this.ctx.canvas.width/this.fullPower, 30);
```

Using a square geometry

We will use the same class, `SquareGeometry`, that we created in Chapter 9, *Ray Casting and Filters*. Let's quickly refresh our memory. Open the `SquareGeomtery.js` file from `client/primitive`. The following code snippet is present in this file:

```
SquareGeometry = inherit(Geometry, function ()
{
  superc(this);
  this.vertices=[
    -1.0,-1.0,0.0,//0
    1.0,-1.0,0.0,//1
    -1.0, 1.0,0.0,//2
    1.0, 1.0,0.0//3
  ];
  var uvs=[ 0.0, 0.0,
            1.0, 0.0,
            0.0, 1.0,1.0,1.0];
  var normal = vec3.fromValues( 0, 0, 1 );
  var face = new Face();
  face.a=0;
  face.b=1;
  face.c=2;
  //...
  var face = new Face();
  face.a=2;
  face.b=1;
  face.c=3;
  //...
});
```

The class first defines four vertices and four texture coordinates. Then, it creates two faces (triangles) using vertex indices (0, 1, 2) for the first triangle and indices (2, 1, 3) for the second triangle.

Implementing the Sprite class

In the `Sprite` class, we initialize our `SquareGeometry` and `SpriteTexture` classes. Open the `Sprite.js` file from `client/primitive/game` in your editor.

The `Sprite` class is a general class and inherits the `StageObject` class:

```
Sprite= inherit(StageObject, function (){
  superc(this);
  this.geometry=new SquareGeometry();
```

It uses the date function to create the unique `textureIndex` and `canvasId` variables:

```
var currentDate=new Date();
  this.textureIndex=currentDate.getMilliseconds();
  this.materialFile=this.textureIndex;
  this.canvasId="c_"+this.textureIndex;
```

For each sprite object, we create a new `canvas` element and add it to our HTML body. Then, we initialize our `SpriteTexture` class by passing the ID for our newly created `canvas` element. Also, note that we have set the `style` attribute of the `canvas` element to `display:none` so that it is not visible to the user, using the following code:

```
jQuery("body").append("<canvas id='"+this.canvasId+"'
style='display:none'></canvas>");
  this.spriteTexture=new SpriteTexture(this.canvasId);
  this.backgroundColor="#ffffff";
  this.color="#333333";
```

Also, as the default scale is (1, 1, 1), we increased the size for our sprite by scaling it using the following line of code:

```
this.scale=vec3.fromValues(10,10,10);
});
```

The following function, `drawText`, in turn invokes the `createTexture` function of the `SpriteTexture` class:

```
Sprite.prototype.drawText=function(text){
  this.spriteTexture.backgroundColor=this.backgroundColor;
  this.spriteTexture.color=this.color;
  this.spriteTexture.createTexture(text);
```

```
  this.spriteTexture.createTexture(text);
}
```

Implementing the ModelSprite class

This class inherits the `Sprite` class. The objective of the class is twofold:

- Set the position of the sprite at a delta distance from the model so that it moves with the model
- Set the orientation of the sprite to face the camera

Open the `ModelSprite.js` file from `client/primitive/game` in your editor. The constructor takes two parameters: the `model` object (the sprite has to be associated with) and the `camera` object:

```
ModelSprite= inherit(Sprite, function (model,cam){
  superc(this);
  this.model=model;
  this.camera=cam;
  this.scale=vec3.fromValues(8,8,8);
  //The relative values to the model position
```

The delta distance from the model is initialized to zero as shown in the following code snippet:

```
this.deltaX=0;
  this.deltaY=0;
  this.deltaZ=0;
});
```

We have overridden the `update` function of the `StageObject` class:

```
ModelSprite.prototype.update=function(){
```

We have discussed on numerous occasions that the model matrix is the inverse of the camera matrix. Hence, we set `this.matrixWorld` to the inverse of the camera matrix, as shown in the following line of code:

```
mat4.invert(this.matrixWorld,this.camera.viewMatrix);
```

Then, we scale our transformation matrix, `this.matrixWorld`, as shown in the following line of code:

```
mat4.scale(this.matrixWorld,this.matrixWorld,this.scale);
```

We then set the position of the sprite by setting the values of *m30*, *m31*, and *m32*

indices of the `this.matrixWorld` array to the values got from adding the model positions and the delta values on the three axes, as shown in the following code:

```
this.matrixWorld[12]=this.model.position[0]+this.deltaX;
this.matrixWorld[13]=this.model.position[1]+this.deltaY;
this.matrixWorld[14]=this.model.position[2]+this.deltaZ;
}
```

Understanding the main flow code

Open the `10-2D-Sprites-And-Text.html` file from the `client` folder in your favorite editor.

We have only changed our `loadStageObject` function to add our `ModelSprite` class. Once our `RiggedMesh` class is loaded, we initialize our `ModelSprite` class with our `RiggedMesh` class' object and the `camera` class' object. We set `deltaY` to render the sprite of 16 units above the model. We then add the texture to the texture array in the `stage` object.

Remember that we store textures and `stageObject` in the main `stage` object. The textures are mapped to `stageObject` with `textureIndex`. We did this so that if multiple objects use the same texture, then we do not load them multiple times, as explained in Chapter 6, *Applying Textures and Simple Animations to Our Scene*. The following code explains the `loadStageObject` function:

```
function loadStageObject(url,location,rotationX,rotationY,rotationZ
){
  //The relevant code changes …....................
  if(data.bones){
    stageObject=new RiggedMesh();
    var sprite=new ModelSprite(stageObject,cam);
    sprite.deltaY=16;
    sprite.drawText("Mr. Red");

stage.addTexture(sprite.textureIndex,sprite.canvasId,document.getEl
ementById(sprite.canvasId),true,true);
    stage.addModel(sprite);

  }
  //The relevant code changes...
}
```

Communicating in real time

Real-time communication is a problem that web developers have been trying to solve for a very long time. The problem is that the client can ping the web server, but the server cannot contact the client. The reason is that the client requests a web page, the server sends the response and the socket is closed. Although the HTTP 1.1 uses persistent connections for multiple requests response, it times out after a specified time. The directive behind a persistent connection is HTTP keep-alive.

This posed a problem in solving real-time communication issues using web servers and HTTP protocols. For example, in a multiplayer game, the information related to the user moving its character needs to be broadcast to all users playing the game. Earlier when we posted the game state to our web server, there was no robust and scalable solution to communicate this information to the other clients in the game room. Numerous techniques evolved to solve this problem. The most obvious solution was that we can write a JavaScript code to request the latest information from the server, using Ajax each second or once in two seconds, which is called **polling**. However, polling is not a scalable solution as it can swamp the web server with an unlimited number of requests. Even if the server did not have any information to offer, the server was pinged continuously by browsers. The only scalable solution was a server push. In this book, we will briefly discuss the two most widely used solutions that can be implemented with most web servers, and they are as follows:

- Long polling
- A new W3C standard—WebSockets

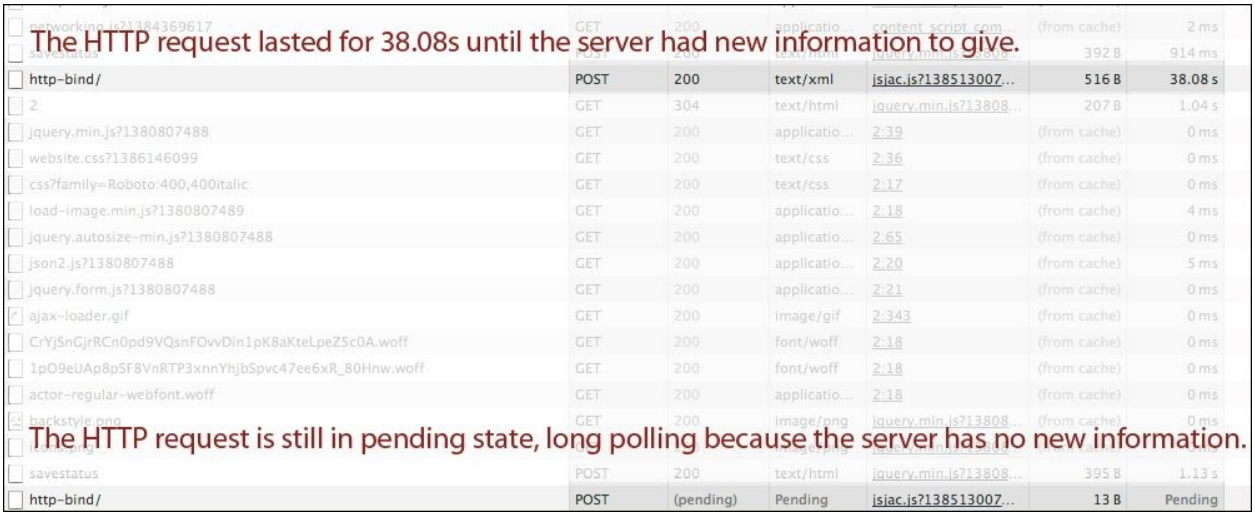
Understanding Ajax long polling

Ajax long polling is the most prominent solution for real-time communication. Let's understand how long polling works:

- After a browser opens a web page, it has a JavaScript code that sends a request to the server.
- The server does not respond immediately and waits until it has some new information to give.
- When the new information is available, the server sends it over a keep-alive session.
- The moment the client receives the information, it sends another request.
- The important point to note is that in HTTP long polling, if the web page wants to request other information, then it opens another connection and does not use the same persistent connection. So, at any time, a maximum of two connections are opened.

The most prominent implementation is in the XMPP protocol in **Bidirectional-streams Over Synchronous HTTP** (**BOSH**), used for chatting over HTTP. The following screenshot demonstrates long polling in a chat system.

The following screenshot shows two HTTP requests. One lasted for 38.08 seconds. Don't worry, the server is not slow but whenever the server had new information, the request was completed. If the server did not have new information, the second request was in the pending state.


The HTTP request lasted for 38.08s until the server had new information to give.
http-bind/ POST 200 text/xml jsjac.js?138513007... 516 B 38.08 s
The HTTP request is still in pending state, long polling because the server has no new information.
savestatus POST 200 text/html jquery.min.js?13808... 395 B 1.13 s
http-bind/ POST (pending) Pending jsjac.js?138513007... 13 B Pending

Understanding WebSockets

WebSocket is a new HTML5 feature for clients to communicate without the overhead of the HTTP protocol. The latest version is RFC 6455 (http://tools.ietf.org/html/rfc6455). It provides us with full duplex bidirectional communication between the server and the client, without using any HTTP methods. It has its own protocol specification and has its own API. It offers the most reliable, robust, scalable, high-performance solution. The beauty is that the communication can happen over the same port as HTTP. A WebSocket server can coexist with an HTTP server on the same port.

Before the communication starts, the client needs to perform a handshake with the server. The client sends a handshake request. The following sample request is an HTTP handshake request for an upgrade to the WebSocket protocol. Remember that the WebSocket server is set up on the same port as the HTTP server; hence, an upgrade request is important. The following is the sample request:

```
GET chat HTTP1.1
Host: server.multiplayergame.com
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Key: lIHhbXBsZSBub25jZQ==
Origin: http://multiplayergame.com
Sec-WebSocket-Protocol: game
Sec-WebSocket-Version: 13
```

The following is the response from the server:

```
HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: game
```

If you have noticed, the server switched protocols from HTTP to WebSocket.

Understanding the WebSocket API

We want to introduce some WebSocket API functions implemented by most browsers. The following code opens a connection over a nonsecure port:

```
var connection = new WebSocket('ws://mygame.org:4000/game')
```

The `wss` protocol is for secure communication:

```
var connection = new WebSocket('wss://mygame.org:4000/game');
```

The second parameter, `['game']`, is a subprotocol to inform the server about the use of the connection:

```
var connection = new WebSocket('wss://mygame.org:4000/game',
['game']);
```

The response handler when the connection is opened is shown in the following code snippet:

```
connection.onopen = function(){
  console.log('Congrats Connection opened!!!');
}
```

The response handler when the connection is closed is shown in the following code snippet:

```
connection.onclose = function(){
  console.log('Sorry Connection closed!!!');
}
```

To explicitly close the connection, use the following line of code:

```
connection.close();
```

The handler to handle any errors in connection is shown in the following code snippet:

```
connection.onerror = function(error){
  console.log('Shit something went bad: ' + error);
}
```

API calls to send and receive messages are shown in the following code snippet:

```
connection.send('Hey?');
connection.onmessage = function(e){
  var data  = e.data;
  console.log(data);
}
```

Sending JSON messages are shown in the following code snippet:

```
var message = {
```

```
  'username': 'sumeet2k06',
  'comment': 'Trying to tell something about WebSockets'
};
connection.send(JSON.stringify(message));
```

Understanding the WebSockets server

A regular HTTP server cannot handle WebSockets requests. We need to install libraries or some external servers to handle requests. There are many WebSocket servers. Two of them are listed as follows:

- phpwebsocket (https://code.google.com/p/phpwebsocket/) for PHP
- Socket.IO (http://socket.io/) for Node.js

As our book uses JavaScript throughout, we would like to work with Node.js and Socket.IO. Also, Socket.IO offers a very powerful client-side JavaScript library, which works even if the browser does not support WebSockets.

Using Node.js and Socket.IO for multiplayer games

Node.js is based on the event-based programming style, where the flow of the application is determined by the events. It is a new generation style of writing a scalable server-side code. Unfortunately, this book is not on Node.js; hence, we would like to completely keep our focus on Node.js-based HTTP servers and Socket.IO. For further reading, you can refer to the following links:

- Installing Node.js (http://howtonode.org/how-to-install-nodejs)
- Loading node modules (http://nodejs.org/api/modules.html)
- Emitter pattern (http://nodejs.org/api/all.html#all_events)
- Stream (http://nodejs.org/api/all.html#all_stream)

Our focus in this book is to discuss two modules of Node.js: HTTP and Socket.IO.

Implementing the HTTP server using Node.js

Node.js comes with a built-in HTTP module. The following sample code listing explains how to work with a node HTTP server.

Note

We are discussing the node HTTP server because once your game page uses WebSockets, we cannot simply click on the HTML page to run a WebSocket client. The WebSocket client page should come from the same domain as the WebSocket server, otherwise it may give security issues. Hence, we need to request the HTML page from a web server running WebSocket of the same port.

The first line will import an `http` module and can also be used to load the module installed by NPM.

Open the `server.js` file in your editor. It contains the following line of code:

```
var http = require('http');
```

The `http` module with the `createServer` function creates a server object:

```
var httpserver = http.createServer();
```

Node.js is completely event-based and an event defines the flow of code. It is basically asynchronous by design. When we start the server, it first creates the HTTP server object and waits for incoming HTTP requests. When it receives a request, the `on` event handler is invoked with the request (`req`) and response(`res`) as the parameters:

```
httpserver.on('request', function(req, res) {
```

After receiving the request, the server sets the HTTP header status code to `200` and sets the content type to plain text:

```
res.writeHead(200, {'Content-Type': 'text/plain'});
```

Then, it writes a string to the response objects:

```
res.write('Hello Game World!');
```

It then flushes the response buffer by invoking the `end` method:

```
res.end(); });
```

The server is set to start at port 4000 by the following line of code:

```
server.listen(4000);
```

Execute the following command to start the server:

```
node server.js
```

Let's walk through another sample code, which reads a file and sends its contents as a response:

```
var fs = require('fs');
var http=require('http')
var server=http.createServer(handler);
server.listen(4000);
function handler(req, res) {
  res.writeHead(200, {'Content-Type': 'video/mp4'});
  var rs = fs.createReadStream('test.mp4');
  rs.pipe(res);
}
```

The preceding code uses another powerful node module, `fs`, which stands for filesystem. The `fs` module offers a wide variety of functions to read and write to file using streams.

The preceding code requests the `fs` module object. It helps us create the file read stream. Then, when the server receives a request, it sets a response header content type to `'video/mp4'`. Node.js offers a very powerful function, `pipe`. This function avoids the slow client problem. If the client is slow, we do not want to fill up our memory with unflushed buffers. So, we pause the read stream and when the consumer has finished reading the data, we resume. We do not want to manage the resuming and pausing of the buffers/streams. Hence, the `pipe` function manages it for you. It is a function of readable stream interface and takes the destination writable stream interfaces as an argument and manages the complete read/write cycle. In our case, the read stream (`rs`) is our file and the write stream is our response (`res`) object (`rs.pipe(res);`).

Understanding Socket.IO

Socket.IO is available as a separate NPM module. It can be installed using the following command:

```
npm install socket.io
```

Socket.IO implements a socket server and also provides a unified client library that does not distinguish between clients, using WebSockets or any other type of mechanism such as flash sockets. It provides a unified API that abstracts away the implementation details of WebSockets.

Let's look at the basic socket server implementation:

```
var io = require('socket.io').listen(400);
io.sockets.on('connection', function (socket) {
  socket.on('any event', function(data) {
    console.log(data);
  });
});
```

The preceding code initializes a basic socket server and waits for requests at port number `4000`. The primary event handler is attached to the connection event. Once the server receives a connection request, it creates a socket object. The socket defines an event on the arbitrary label, `any event`. When the client emits an event with the `any event` label, this handler (`function (socket)`) is invoked. The label is just created to differentiate different kinds of messages that a client sends.

Let's look at a basic Socket.IO client's code:

```
<html>
<head>
<title>Socket.IO example application</title>
<scriptsrc="http://localhost:400/js/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost:4000');
  socket.emit('any event', 'Hello Game world.');
</script>
</head>
<body></body>
</html>
```

In the preceding code, we have included `js/socket.io.js`. In the code, we first open a connection by invoking the `io.connect` function and then invoke the `socket.emit` function with the `any event` label. Note that we have not used any WebSocket API calls in the client code as the Socket.IO client library abstracts the complete implementation.

Let's say that the preceding code was saved in `index.html`, but when we simply double-click on the file, our code might not run as the WebSocket API can generate security issues with local files. Hence, we would want this file to be rendered by our web server. So, let's combine our WebServer and socket server to handle both types of requests with the help of the following code:

```
var httpd = require('http').createServer(httphandler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');
httpd.listen(400);
function httphandler(req, res) {
  fs.readFile(__dirname + '/index.html', function(err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading File');
    }
    res.writeHead(200);
      res.end(data);
    }
  );
}
io.sockets.on('connection', function (socket) {
  socket.on('clientData', function(data) {
    socket.emit('serverData', data);
    socket.broadcast.emit('serverData', socket.id +content);
  });
});
```

The preceding code first creates an HTTP server object. It then creates a socket server object and binds itself to the HTTP server port (`require('socket.io').listen(httpd);`). Now, the socket and web servers are listening at the same port number.

The HTTP server defines a request handler, `httphandler`, and reads the file `index.html`. If an error occurs while reading, then it sets the HTTP response status header to `500` and sets response content to an error message. If no error occurs, it writes the file content to the response.

The socket server implementation introduces us to a new server-side function, `socket.broadcast.emit('serverData', socket.id +content)`. This server-side function sends the data to all the clients connected to the socket server (except the socket it is fired on) with the client event, `serverData`. Let's just understand the sequence of events.

The following client code defines the client handler function for the server event, `serverData`. It simply logs the received data. When the *Enter* key is pressed, it sends the data to the socket with the `clientData` event:

```
var socket = io.connect('http://localhost:400');
  socket.on('serverData', function(content) {
  console.log(content);
}
var inputElement = document.getElementById('input');
inputElement.onkeydown = function(keyboardEvent) {
  if (keyboardEvent.keyCode === 13) {
    socket.emit('clientData, inputElement.value);
    inputElement.value = '';
    return false;
  } else {
    return true;
  }
};
```

The corresponding server handler function for the client event, `clientData`, on invocation, sends the data back to the client and broadcasts the message. Then, the code line sends the data back to the client and invokes its `serverData` event. This code broadcasts the message to all the clients connected to the socket server except the socket it is fired on. As we can see in the following code listing, we first emit the `serverData` event on the calling socket and then broadcast `serverMessage` to all sockets:

```
io.sockets.on('connection', function (socket) {

  socket.on('clientData', function(data) {

    socket.emit('serverData', data);
    socket.broadcast.emit('serverMessage', socket.id +content);
  });
});
```

As you can see in the preceding code, the client and server both use the `emit`

function to invoke each other's events. The other important aspect is `socket.id`, which is a unique key that identifies each client. The complete implementation of `socket.id` is managed by the Socket.IO library.

Learning the Socket.IO API

The Socket.IO API consists of server-side functions and client-side functions. The API has a list of reserved events. The API defines two objects on the server side: `io.sockets` and `socket`. They are explained as follows:

- `io.sockets`: This has only one reserved event connection.

  ```
  io.sockets.on('connection', function(socket) {})
  ```

  The preceding event is generated when a new connection is requested from the client side.
- `socket`: This offers two reserved events, `message` and `disconnect`.

  The `message` event is fired when the `send` function of the socket object is used to emit events.

  The client code is as follows:

  ```
  <script>
  var socket = io.connect('http://localhost/');
  socket.on('connect', function () {
    socket.send('hi');
    socket.on('message', function (msg) {
      });
    });
  </script>
  ```

  The server code is as follows:

  ```
  var io = require('socket.io').listen(80);
  io.sockets.on('connection', function (socket) {
    socket.on('message', function () { });
    socket.on('disconnect', function () { });
  });
  ```

  In the preceding server code, we handle two events, `message` and `disconnect`. The `disconnect` event is fired when the client-server connection is closed.

The client socket's reserved events

The client socket has many reserved events defined. Refer to the following list:

- `connect`: This is emitted when the socket connection is successful.
- `connecting`: This is emitted when the socket is attempting to connect with the server.
- `disconnect`: This is emitted when the socket is disconnected.
- `connect_failed`: This is emitted when Socket.IO fails to establish a connection to the server and has no other transport to fallback on.
- `error`: This is emitted when an error occurs and it cannot be handled by the other event types.
- `message`: This is emitted when a message sent with the `socket.send` function is received.
- `reconnect_failed`: This is emitted when Socket.IO fails to reestablish a working connection after the connection was dropped.
- `reconnect`: This is emitted when Socket.IO has successfully reconnected to the server.
- `reconnecting`: This is emitted when the socket is attempting to reconnect with the server.

Understanding Socket.IO rooms

Rooms allow us to partition the connected clients. Rooms allow events to be emitted to subsets of the connected client list.

Leaving and joining

A user can simply leave or join the room by invoking the following socket functions:

```
socket.join('room')
socket.leave('room')
```

If the room does not exist, a new user joins the room, and the room is created. When the last user leaves the room, then the room is automatically pruned. You do not need to leave the room at a disconnected event.

Broadcasting an event in the room

The following are the two ways to broadcast a message to all room participants:

- We have a `socket` object of one of the clients in the room:

  ```
  socket.broadcast.to('room').emit('event name', data)
  ```

This function will not send the data back to the emitting client/socket.

- We can also use the `io.sockets` object directly:

```
io.sockets.in('room').emit('event_name', data)
```

This will emit an event, `event_name`, to all participants in the room.

Getting the room's information

Socket.IO provides us with a powerful set of functions to get the room's information. They are as follows:

- `io.sockets.manager.rooms`: This is used to get the list of rooms and socket IDs. It returns a hash with the name of the room as the key and the list of socket IDs as value.
- `io.sockets.clients('room')`: This will return the array of sockets of the connected client.
- `io.sockets.manager.roomClients[socket.id]`: This returns the dictionary of rooms that a particular client socket has joined.

The name of the room that is returned from all the preceding API calls has a leading `/` appended to it. It is used by Socket.IO internally, which we should eliminate.

Storing user data on the server side

There are many cases where we need to store the user data for a session. We will lose the data once the user disconnects. The following code uses Socket.IO API's `get` and `set` functions of the server socket class to store and retrieve the user data:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
  socket.on('set nickname', function (name) {
    socket.set('username', name, function () {
      socket.emit('ready');
    });
  });

  socket.on('msg', function () {
    socket.get('username', function (err, name) {
      console.log('Chat message by ', name);
```

```
    });
  });
});
```

The following code sets the `username` variable:

```
// Sets the username variable.
socket.set('username', name);
```

The following code retrieves the stored `username` variable:

```
socket.get('username', function (err, name) {
  console.log('Chat message by ', name);
    });
```

Implementing a multiplayer game

The objective of this section is to implement what we have learned in the previous sections. We simply want to implement a spectator player, a player who simply launches our game can only view it. This simple implementation is a good start for the complete multiplayer game implementation.

Let's first understand the data that we need to communicate to our spectator client. In our game, currently we have nearly all static assets, except camera, bullets, and grenade. Also, the location/direction of bullet and grenade are derived from the camera location. So, all we need to communicate to the other client is:

- Camera position, direction, left and up vectors
- The bullet fire event and the bullet index that was fired
- The grenade fire event

As we develop the complete game, we will need to communicate more information such as dynamic model locations and scores. However, for the purpose of learning, we will stick to the preceding information.

Also in Socket.IO, we only transfer the string data from the client to the server. Hence, we will use JSON strings for communication. Let's take a look at the structure of JSON objects we will use to communicate between clients.

In the following screenshot, we have described the information we need to transfer in the form of a JSON object:

Camera Object (Position,direction,up,left)

- JSON
  - username : "you"
  - roomname : "ourgame"
  - message
    - left
      - 0 : -1
      - 1 : 0
      - 2 : 0
    - up
      - 0 : 0
      - 1 : 0.9695636630058289
      - 2 : -0.24483931064605713
    - dir
      - 0 : 0
      - 1 : -0.24483931064605713
      - 2 : -0.9695636630058289
    - position
      - 0 : 290
      - 1 : 25
      - 2 : 100
    - type : "camera"

Bullet Object (Bullet Index)

- JSON
  - username : "you"
  - roomname : "ourgame"
  - message
    - type : "bullet"
    - bulletIndex : 0

Grenade Object

- JSON
  - username : "you"
  - roomname : "ourgame"
  - message
    - type : "grenade"

Understanding events and the code flow

In the preceding section, we have described the data that we need to transfer and in this section, we will list the custom events that we will emit for the preceding communication and also list the reserved events. We describe in the form of an algorithm for both the client and server sides. The sequence defines the event flow from the server to the client and vice versa. The following steps are basic initialization steps. They describe the sequence from the user connect request until the user room join event:

- The server creates a client socket and emits a `login` event using the following line of code:

  ```
  socket.emit('login');
  ```

- The client handles `.on('login',handler)` and emits the server `login` custom event and sends its username using the following line of code:

  ```
  socket.emit('login', "player");
  ```

- The server handles `.on('login',handler)`, saves the username in private user data and sends the client the confirmation that the user has logged in by emitting the `loggedIn` event. It broadcasts the event with the actual username to all clients and emits with the username `you` to the emitting client, using the following lines of code:

  ```
  socket.broadcast.emit('loggedIn', JSON.stringify(message));
  message.username="you";
  socket.emit('loggedIn', JSON.stringify(message));
  ```

- The client handles the `loggedIn` custom event. In our code, we are not listing all users but we can use it to create a list of connected users. In our case, if the emitted username is `you`, then the socket emits the `join` event to join the `ourgame` room, using the following line of code:

  ```
  socket.emit('join', "ourgame");
  ```

- The server handles the `join` event by adding the emitting socket to the room using the following line of code:

  ```
  socket.join(room);
  ```

  Then, it emits the emitting socket and the `roomJoined` event to all users in the room by using the following line of code:

```
socket.emit('roomJoined', JSON.stringify(message));
socket.broadcast.to(room).emit('roomJoined',
JSON.stringify(message));
```

- The client handles the `roomJoined` event. We can use it to update the user in the room list.

  Once the user has joined the room, the next steps are iterative. This means that both the server and client wait until a player performs an action such as move or fire a grenade. The following steps explain the event flow post the user performs any key or mouse event.
- The client user moves the camera or fires a bullet or grenade. The code generates the respective JSON object and emits a `clientMessage` event.
- The server handles the `clientMessage` event, creates a new JSON object with username, room name, and populates the message with the received JSON object and emits the `serverMessage` event with the data back to all users in the room and emitting client with the username `you`.
- The client handles the `serverMessage` event, updates the camera location, or fires the bullet/grenade only if the username is not `you`. We send the message back to the emitting client with the username `you`, even though no action has to be taken as an acknowledgment of the emitted `clientMessage` event.

The code walkthrough

Now let's dive deep into the code and understand the complete client-server communication.

Note

We have moved our complete JavaScript, HTML, and CSS in the `client` folder inside the code files of this chapter to be rendered using the web server.

The server code

Open the `serverhttp.js` file in your editor. The server code starts with initialization of the web server to render the client code, and we also attach our socket server to the HTTP server, as follows:

```
var http = require('http').createServer(handler),
url = require('url'),
path = require('path'),
fs = require('fs');
var io = require('socket.io').listen(http);
```

We list the mime types as follows:

```
var mimeTypes = {
  "html": "text/html",
  "jpeg": "image/jpeg",
  "jpg": "image/jpeg",
  "png": "image/png",
  "js": "text/javascript",
  "json": "text/javascript",
  "css": "text/css"};
```

The HTTP request handler reads the request URL (for example, `http://127.0.0.1/10-Multiple-Player-Programming-Spectator.html`) and retrieves the path name (for example, `10-Multiple-Player-Programming-Spectator.html`). To retrieve the physical path, it appends the path to the physical path (`process.cwd()`) concatenated with the `/client` folder. We concatenate the `/client` folder name as we want to render only files of that folder. We use path modules function, `join`, to generate the physical path and use the reader stream `pipe` function to render the file. We also set the content type based on the file extension and retrieve the corresponding mime type from the `mimeTypes` array. The following code is of the `handler` function:

```
function handler(req, res) {
  var uri = url.parse(req.url).pathname;
  var filename = path.join(process.cwd(),'client', uri);
  path.exists(filename, function(exists) {
    if(!exists) {
      console.log("not exists: " + filename);
      res.writeHead(200, {'Content-Type': 'textplain'});
      res.write('404 Not Found\n');
      res.end();
      return;
    }
    var mimeType = mimeTypes[path.extname(filename).split(".")[1]];
    res.writeHead(200, {'Content-Type':mimeType});
    var fileStream = fs.createReadStream(filename);
    fileStream.pipe(res);

  }); //end path.exists
}
http.listen(80);
```

The socket server waits for the client request:

```
io.sockets.on('connection', function (socket) {
```

On a completed client request, it emits a `login` event to emit the client:

```
socket.emit('login');
```

On a `login` request from the client, it stores the username in the `username` session variable using the socket API function `set` and emits the `loggedIn` event to all users and the emitting client with the username `you`, as shown in the following code:

```
socket.on('login', function(username) { socket.set('username',
username, function(err) {
  if (err) { throw err; }
  var message={};
  //message.key="logged";
  message.username=username;
  socket.broadcast.emit('loggedIn', JSON.stringify(message));
  message.username="you";
  socket.emit('loggedIn', JSON.stringify(message));
});
});
```

On the join request, it sets the user session variable, `room`, with a room name so that it can be used later. In case the user wants to join another room, we save the room name to remove the user from the existing room. Hence, we first remove the user from the existing room and add it to the new room. Then, the code emits the `roomJoined` event to all users of the room and the emitting client, as shown in the following code:

```
socket.on('join', function(room) { socket.get('room', function(err,
oldRoom) {
  if (err) { throw err; }
  socket.set('room', room, function(err) {
    if (err) { throw err; }
    socket.join(room); if (oldRoom) {
      socket.leave(oldRoom);
    }
    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      var message={};
      //message.key="join";
      message.username=username;
      socket.emit('roomJoined', JSON.stringify(message));
      socket.broadcast.to(room).emit('roomJoined',
JSON.stringify(message));
    });
  });
});
});
```

On the `clientMessage` event, the handler first retrieves the `username` variable. In case `username` cannot be retrieved, it sets `socket.id` as the `username` variable. Then, it creates a new JSON object message and assigns username, room name, and received JSON object to the `message` property. It emits `serverMessage` to the emitting client as well as all users in the room, as shown in the following code:

```
socket.on('clientMessage', function(content) {

  socket.get('username', function(err, username) { if (! username)
{
    username = socket.id;
  }
  socket.get('room', function(err, room) {
    if (err) {
```

```
      throw err;
    }
  var broadcast = socket.broadcast;

  if (room) {
    broadcast.to(room);
  }
  var message={};
  message.username=username;
  message.roomname=room;
  message.message=JSON.parse(content);
  broadcast.emit('serverMessage', JSON.stringify(message));
  message.username="you";
  socket.emit('serverMessage', JSON.stringify(message)); });

});
});

});
```

The client code

Now, we will walk through the client code. Open `10-Multiple-Player-Programming.html` from the `client` folder in your editor. The client code connects to the server on the page load. This following code is present in this file:

```
<script type="text/javascript">
var socket = io.connect('http://127.0.0.1');
```

When the connection is created, the server emits a `login` event. The client responds to the server with the username `player` for the playing user, and the username `spectator` for the other user, as shown in the following code:

```
socket.on('login', function() {
  socket.emit('login', "player");
});
```

The client responds to the logged-in server event and emits the `join` event with the room name `ourgame` as a parameter, as shown in the following code:

```
socket.on('loggedIn', function(content) {
  console.log(content);
  var messageObj=JSON.parse(content);
  if(messageObj.username=="you")
  socket.emit('join', "ourgame");
```

```
});
socket.on('roomJoined', function(content) {
  console.log(content);
});
```

On `serverMessage`, the client invokes the `applyData` function. The `applyData` function parses the response and checks the `messageObj.username` parameter. If the value is not `you`, it updates the game state. Depending on the `messageObj.message.type` value, it performs the desired action. If the value is `camera`, then it updates the `cam` object with received values. The `applyData` function is explained in the following code:

```
socket.on('serverMessage', function(content) {
  applyData(content);
});

function applyData(message) {
  try{
    console.log(message);
    var messageObj=JSON.parse(message);
    if(messageObj.username!="you"){
      switch(messageObj.message.type)
      {
        case "camera":
cam.left=vec3.fromValues(parseInt(messageObj.message.left[0]),parse
Int(messageObj.message.left[1]),parseInt(messageObj.message.left[2]
));

cam.up=vec3.fromValues(parseInt(messageObj.message.up[0]),parseInt(
messageObj.message.up[1]),parseInt(messageObj.message.up[2]));

cam.dir=vec3.fromValues(parseInt(messageObj.message.dir[0]),parseIn
t(messageObj.message.dir[1]),parseInt(messageObj.message.dir[2]));
        cam.pos=vec3.fromValues(
          parseInt(messageObj.message.position[0]),
          parseInt(messageObj.message.position[1]),
          parseInt(messageObj.message.position[2]));
        break;
        case "grenade":
          leftHand.visible=true;

        break;
        case "bullet":

bullets[parseInt(messageObj.message.bulletIndex)].initialize();
        break;
```

```
      }
    }else{
      console.log(messageObj.message.type);
    }catch(e){
    console.log(e);
  }
  console.log(message);
}
```

The following `sendRoomMessage` function is invoked from different sections of our application, whenever we want to update the game state:

```
function sendRoomMessage(message){
  socket.emit('clientMessage', message);
}
</script>
```

Now, let's look at other modifications in the code to update the game state. The first modification is in the `start` function. The `start` function assigns a `cameraChangedCallBack` handler function in `keyboardInteractor`. The following callback function is invoked when the camera location is changed:

```
function start() {
...
  keyboardInteractor.cameraChangedCallBack=cameraLocationChanged;
...
}
```

The following callback function, `cameraLocationChanged`, receives the camera location and sets the message type to `"camera"`. It then invokes the `sendRoomMessage` function:

```
function cameraLocationChanged(cameraLocationStatus){
  cameraLocationStatus.type="camera";
  sendRoomMessage(JSON.stringify(cameraLocationStatus));
}
```

The following `handleKeys` function invokes the `sendRoomMessage` function on both events of throwing the grenade and firing a bullet with the corresponding `message.type` value:

```
function handleKeys(event) {
  switch(event.keyCode) {//determine the key pressed
    case 13:
      leftHand.visible=true;
      var message=[];
```

```
      var message={};
      message.type="grenade";
      sendRoomMessage(JSON.stringify(message));
    break;
    case 32://s key
      bullets[nextBulletIndex].initialize();
      var message={};
      message.type="bullet";
      message.bulletIndex=nextBulletIndex;
      sendRoomMessage(JSON.stringify(message));
...
    }
    break;
  }
}
```

Open `KeyBoardInteractor.js` from `client/primitive` in your editor. In this class, we have added a `cameraChangedCallBack` delegate to handle the call back when the location/direction of camera changes The `KeyBoardInteractor` function is as follows:

```
function KeyBoardInteractor(camera,canvas){
...
  this.cameraChanged=false;
  this.cameraChangedCallBack=null;
...
}
```

If the location of the camera changes, the `handleKeys` function updates the status variable `cameraChanged` to `true`, on each key event. We check whether the `cameraChanged` variable is `true` and whether the handler function has been assigned, then we create a camera status object with the up, left, position, and direction vectors. We then invoke the handler function and pass `cameraStatus` as a parameter. We also update the `cameraChanged` variable to `false`. The `handleKeys` function is as follows:

```
KeyBoardInteractor.prototype.handleKeys=function (event) {
  if(event.shiftKey) {
    switch(event.keyCode) {//determine the key pressed
      case 65://a key
        this.cam.roll(-Math.PI * 0.025);//tilt to the left
        this.cameraChanged=true;
      break;
...
  }
...
```

```
...
  if(this.cameraChanged==true&&this.cameraChangedCallBack!=null){
    var cameraStatus={};
    cameraStatus.left=this.cam.left;
    cameraStatus.up=this.cam.up;
    cameraStatus.dir=this.cam.dir;
    cameraStatus.position=this.cam.pos;
    this.cameraChangedCallBack(cameraStatus);
    this.cameraChanged=false;
  }

}
```

Summary

In this chapter, we first covered the implementation of the canvas 2D context as texture in our game scene. This is a very powerful tool, not just limited to game development, but is also used in many 3D applications such as product customization tools. The canvas 2D API offers a very powerful list of functions to manipulate images using its pixel data. This gives us the ability to add dynamic textures to our game scene.

In the second section of the chapter, we introduced you to multiplayer games and used WebSockets to create a sample spectator client. We used the module Sockets.IO of Node.js, to implement our multiplayer game.

In this book, we have trained ourselves in topics such as rendering, animation, physics engines, collision detection, bone animations, and multiplayer games to evolve our world in 5000 A.D. Now, we leave it up to you to complete your world and add components that excite the users of your game.

Index

A

- addBone function / Implementing the RiggedMesh class
- add function / Loading the animation data
- addModel function / Implementing Stage.js, Using a rigid body (collider) for each scene object, Changing the color of the selected object
- addStageObject function / Understanding the main code
- addToUpdate function / Loading the animation data
- affine transformations / Classifying into linear and affine transformations
- Ajax long polling
  - about / Understanding Ajax long polling
- ambient component / Understanding the illumination/reflection model
- ambient lighting
  - about / Ambient lighting
- Ammo
  - about / Ammo.js
- Ammo.js
  - about / Ammo.js
- angVel (angular velocity) / Implementing the free camera
- animate() function / Adding gravity and a rigid body to the game scene
- animate function / Understanding the main code
- animation, types
  - time-based animation / Understanding time-based animation, Implementing time-based animation
  - frame-based animation / Understanding frame-based animation
- Animation class / Loading the animation data
- animation data
  - about / JSON model – animation data
  - loading / Loading the animation data
- AnimationHandler / Loading the animation data
- AnimationHandler class / Loading the animation data
- applyData function / The client code
- apply function / Implementing the basic camera
- aTextureCoord / The vertex shader
- attribute-qualified variables
  - about / Attributes
- aVertexColor attribute / Coloring using the vertex color

- aVertexPosition attribute / Vertex transformation with a basic vertex shader
- Azimuth parameter / Comprehending the orbit camera

B

- basic camera
  - about / Using the basic camera
  - implementing / Implementing the basic camera
- Bidirectional-streams Over Synchronous HTTP (BOSH) / Understanding Ajax long polling
- Bidirectional Reflectance Distribution Functions (BRDFs) / Understanding the illumination/reflection model
- bilinear filtering / Bilinear filtering with mipmapping
- bindBuffer() function / Vertex buffer objects – uploading data to GPU
- bindFramebuffer function / Rendering to framebuffers
- binding
  - about / Simple skinning
- binding matrix / The binding matrix
- bindPostProcessShaderAttributes() function / Implementing the filter
- Blender
  - models, exporting from / Exporting models from Blender
- Blender add-on
  - URL, for downloading / Exporting from Blender
- Blinn-Phong model
  - about / Understanding the illumination/reflection model, The Blinn-Phong model
- Blinn Phong reflection
  - about / Implementing Gouraud shading – Blinn-Phong reflection, Implementing Phong shading – Blinn-Phong reflection
- bone class
  - implementing / Implementing the bone class
- boneGlobalMatrices parameter / Loading the skinned model
- bones
  - about / Comprehending the joint hierarchy
- Box.json
  - Box.obj, converting to / Converting Box.obj to Box.json
- Box.obj
  - converting, to Box.json / Converting Box.obj to Box.json
- Box.obj file, elements
  - mtllib / Exporting models from Blender
  - o / Exporting models from Blender
  - v / Exporting models from Blender

    - vt / Exporting models from Blender
    - vn / Exporting models from Blender
    - s / Exporting models from Blender
    - f / Exporting models from Blender
- Box2dweb
    - about / Box2dweb
- bufferData() function / Drawing using index buffer objects
- buffer objects
    - associating, with shader attributes / Associating buffer objects with shader attributes
- bullet
    - about / Simple bullet action – linear animation
    - objects, reusing in multiple bullets / Reusing objects in multiple bullets
- bullet action / Cheating in the bullet action

C

- calculateMvMatrix function / Implementing the control code
- calculateNormals(vertices,indices) function / Implementing Gouraud shading on a Lambertian reflection model
- calculateVertexNormals function / Calculating normals from vertices and indices, Implementing Geometry.js
- cameraChangedCallBack handler function / The client code
- camera rotation
  - about / Understanding the camera rotation
- camera transformation matrix
  - about / Understanding the view transformation, Understanding the camera matrix
  - components / Comprehending the components of a camera matrix
  - and view matrix, conversions / Converting between the camera matrix and view matrix
- canvas 2D
  - about / Understanding canvas 2D basics and the drawing API
  - URL / Understanding canvas 2D basics and the drawing API
  - using, for textures / Using canvas 2D for textures
- canvas element / Understanding canvas 2D basics and the drawing API
- changes
  - summarizing / Implementing Stage.js
- characters skeleton
  - about / Understanding the basics of a character's skeleton
  - joint hierarchy / Comprehending the joint hierarchy
  - kinematics / Understanding forward kinematics
- CLAMP_TO_EDGE, wrapping mode / Texture wrapping
- clientMessage event / Understanding events and the code flow, The server code
- client socket, reserved events
  - about / The client socket's reserved events
  - connect / The client socket's reserved events
  - connecting / The client socket's reserved events
  - disconnect / The client socket's reserved events
  - connect_failed / The client socket's reserved events
  - error / The client socket's reserved events
  - message / The client socket's reserved events
  - reconnect_failed / The client socket's reserved events

    - reconnect / The client socket's reserved events
    - reconnecting / The client socket's reserved events
- clone class / Implementing StageObject.js
- clone function / Implementing Geometry.js, Implementing StageObject.js
- cloneObjects function / Loading the scene
- closestDistance parameter / Implementing the orbit camera
- collision detection
    - about / Diving deep into collision detection
- CollisionInfo object / Diving deep into collision detection
- color information
    - storing, texture object created / Creating a texture object to store color information
- colors
    - about / Understanding colors, Learning more about colors
    - square, coloring / Coloring our square
    - vertex color, used for coloring / Coloring using the vertex color
- connect / The client socket's reserved events
- connecting / The client socket's reserved events
- connect_failed / The client socket's reserved events
- console.log() / Debugging a WebGL application
- control code
    - implementing / Implementing the control code
    - about / Working with the control code
- Cook-Torrance model
    - about / Understanding the illumination/reflection model
- createBuffer() function / Vertex buffer objects – uploading data to GPU
- createBuffers function / Implementing StageObject.js, Implementing Stage.js
- createFrameBuffer() function / Implementing the filter
- createMultilineText function / Using the sprite texture
- createServer function / Implementing the HTTP server using Node.js
- createShader() function / Compiling and linking shaders
- createTexture() function / The fragment shader
- createTexture function / Using the sprite texture, Implementing the Sprite class
- ctx.measureText(txt).width function / Understanding canvas 2D basics and the drawing API
- ctx.restore() function / Understanding canvas 2D basics and the drawing API

- ctx.store() function / Understanding canvas 2D basics and the drawing API
- ctx.transform()function / Understanding canvas 2D basics and the drawing API
- cubemaps
  - about / Understanding cubemaps and multi-texturing
  - coordinates / Cubemap coordinates
  - loading / Loading cubemaps
  - shader code / Understanding the shader code
- currentText variable / Using the sprite texture

D

- 2D context
  - URL / Understanding canvas 2D basics and the drawing API
- 2D textures
  - about / Texturing basics, Understanding 2D textures and texture mapping
  - and texture mapping / Understanding 2D textures and texture mapping
  - adding, as model labels / Adding 2D textures as model labels
  - sprite texture, using / Using the sprite texture
  - square geometry, using / Using a square geometry
- 3D graphics
  - indices, using to save memory / Using indices to save memory
- 3D mathematics
  - about / Understanding basic 3D mathematics
  - vectors / Vectors
  - matrices / Matrices
  - transformations / Understanding transformations
- 3D objects
  - rendering, transformations for / Understanding transformations required to render 3D objects
  - rendering / Rendering 3D objects
  - exporting, from Blender / Exporting a 3D object from Blender
- 3D software
  - models, exporting from / Exporting models from 3D software in JSON
- 3D textures
  - about / Texturing basics
- diffuse component / Understanding the illumination/reflection model
- Diffuse reflection / Diffuse reflection (Lambertian reflection)
- directional light
  - about / Directional lights
- directionalLightWeighting1 float data type / The fragment shader
- disconnect / The client socket's reserved events
- display memory / Framebuffers
- distance dropdown / Applying filtering modes
- DOFs
  - about / Understanding the basics of a character's skeleton
  - URL / Understanding the basics of a character's skeleton
- Down arrow / Understanding the free camera

- draw() function / Implementing the filter
- drawArray() function / Drawing our primitive
- drawArrays() function / Drawing our primitive, Drawing using vertex buffer objects
- drawArrays function / Drawing our primitive
- drawElements function / Drawing our primitive
- drawScene() function / Drawing using vertex buffer objects, Coloring using the vertex color, Implementing the filter
- drawScene function / Understanding the main code, Using mvMatrix states, Understanding request animation frames, Loading the scene, The fragment shader, Applying filtering modes, Implementing the control code, Using linear interpolation for left-hand rotation, Revisiting the grenade and bullet actions, Loading the skinned model, Changing the color of the selected object
- drawScenePostProcess() function / Implementing the filter
- DYNAMIC_DRAW / Vertex buffer objects – uploading data to GPU

E

- e.clientX attribute / Calculating the screen coordinates of a click
- Elevation parameter / Comprehending the orbit camera
- emitter pattern
  - URL / Using Node.js and Socket.IO for multiplayer games
- error / The client socket's reserved events
- explosionCallBack function / Using texture animation for an explosion effect
- explosion effect
  - texture animation, using for / Using texture animation for an explosion effect

F

- f, Box.obj file element / Exporting models from Blender
- Face.js
  - implementing / Implementing Face.js
- Face class / Implementing StageObject.js
- face object / Understanding a simple terrain – plane geometry
- faces
  - about / Supporting multiple objects
- faceVertexUvs array / Understanding a simple terrain – plane geometry
- FarthestDistance parameter / Implementing the orbit camera
- field of view (FOV) / Implementing Gouraud shading on a Lambertian reflection model, Understanding the viewing frustum
- fillstyle property / Understanding canvas 2D basics and the drawing API
- filter
  - applying, framebuffers used / Applying filters using framebuffers
  - vertex shader / The vertex shader
  - fragment shader / The fragment shader
  - shaders, loading / Loading and linking shaders
  - shaders, linking / Loading and linking shaders
  - implementing / Implementing the filter
- filtering modes
  - applying / Applying filtering modes
- final normal transformation / The final normal transformation
- final vertex transformation / The final vertex transformation
- first-person camera
  - about / Using first-person camera
  - adding / Adding the first-person camera
  - code, improving / Improving the first-person camera code
- flat shading
  - about / Flat shading
- format parameter / Loading textures
- forward kinematics
  - about / Understanding forward kinematics
- fragment shader
  - about / The fragment shader, The fragment shader, The fragment shader, The fragment shader / The fragment shader
- fragment shader code

    - implementing / Implementing the fragment shader code
- frame-based animation / Understanding frame-based animation
- framebuffer object / Implementing the filter
- framebuffer object (FBO) / Offscreen rendering using framebuffers
- framebuffers
    - offscreen rendering / Offscreen rendering using framebuffers
    - rendering to / Rendering to framebuffers
    - used, for applying filters / Applying filters using framebuffers
- Framebuffers
    - about / Framebuffers
- frameTexture object / Implementing the filter
- free camera
    - about / Understanding the free camera
    - implementing / Implementing the free camera
    - using / Using our free camera
    - control code, implementing / Implementing the control code
- FreeCamera class / Understanding the free camera

G

- game engine
  - WebGL, differentiating from / Differentiating WebGL from the game engine
- game scene
  - gravity, adding / Adding gravity and a rigid body to the game scene
  - rigid body, adding / Adding gravity and a rigid body to the game scene
- Geometry.js
  - implementing / Implementing Geometry.js
- Geometry object / Implementing parseJSON.js
  - about / Supporting multiple objects
  - vertices / Supporting multiple objects
  - normals / Supporting multiple objects
  - UV map / Supporting multiple objects
  - per vertex colors / Supporting multiple objects
  - materials / Supporting multiple objects
  - indices / Supporting multiple objects
  - faces / Supporting multiple objects
  - changes / Changes in our Geometry object
- getAttribLocation function / Attributes
- getBoneMatrix function / Loading the skinned model
- getContext method / Understanding canvas 2D basics and the drawing API
- getDistance function / Implementing the orbit camera
- get function / Loading the animation data
- Gimbal lock / Using quaternions
- gl.bufferData() API call / Vertex buffer objects – uploading data to GPU
- gl.pixelStorei function / Loading textures
- gl.pixelstorei function / Loading textures
- gl.texImage2D function / Implementing the filter
- gl.texParameteri function / Loading textures
- gl.uniformMatrix4fv() function / Drawing using vertex buffer objects
- gl.vertexAttribPointer() function / Drawing using vertex buffer objects
- glMatrix
  - URL / Understanding basic 3D mathematics, Understanding ModelView transformations
- glVertexAttribPointer() function, parameters
  - index / Associating buffer objects with shader attributes
  - size / Associating buffer objects with shader attributes

    - type / Associating buffer objects with shader attributes
    - norm / Associating buffer objects with shader attributes
    - stride / Associating buffer objects with shader attributes
    - offset / Associating buffer objects with shader attributes
- gl_FragColor variable / The fragment shader, Coloring using the vertex color
- gl_Position variable / Vertex transformation with a basic vertex shader
- goCloser function / Implementing the orbit camera
- goFarther function / Implementing the orbit camera
- Gouraud shading
    - about / Gouraud shading
    - implementing, on Lambertion reflection model / Implementing Gouraud shading on a Lambertian reflection model
    - implementing / Implementing Gouraud shading – Blinn-Phong reflection
- grenade
    - about / Revisiting the grenade and bullet actions
- Grenade class / Using linear interpolation for left-hand rotation

H

- handleKeys function / The client code
- height maps
  - implementing / Implementing height maps
- http module / Implementing the HTTP server using Node.js
- HTTP server
  - implementing, Node.js used / Implementing the HTTP server using Node.js

I

- index buffer objects
  - about / Index buffer objects
  - used, for drawing / Drawing using index buffer objects
- index parameter / Associating buffer objects with shader attributes
- indices
  - about / Supporting multiple objects
- inherit function / Implementing the free camera
- initBuffer() function / Coloring using the vertex color
- initBuffer() function / Drawing using index buffer objects
- initBuffers() function / Drawing using vertex buffer objects, Implementing Gouraud shading on a Lambertian reflection model
- initBuffers function / Implementing StageObject.js, Understanding 2D textures and texture mapping, Applying a texture to the square
- initData function / Loading the animation data
- initialize function / Using linear interpolation for left-hand rotation
- initializePhysics function / Adding gravity and a rigid body to the game scene
- initializePosition function / Cheating in the bullet action
- initScene() function / Loading the JSON model
- initShaders() function / Compiling and linking shaders, Loading the JSON model, Working with the control code
- initShaders function
  - about / Understanding the main code

  / Understanding the main code, Loading the skinned model
- init_jiglib() function / Adding gravity and a rigid body to the game scene
- internalFormat parameter / Loading textures
- interpolation
  - linear interpolation / Linear interpolation
  - polynomial interpolation / Polynomial interpolation
  - spline interpolation / Spline interpolation
- intersection
  - checking for / Checking for an intersection
- io.sockets / Learning the Socket.IO API
- io.sockets.clients('room') / Getting the room's information
- io.sockets.manager.roomClients[socket.id] / Getting the room's information
- io.sockets.manager.rooms / Getting the room's information
- iSpecular variable / The fragment shader

- isPicked property / Changing the color of the selected object
- ITerrain interface / Understanding the physics shapes, Extending our terrain with physics

J

- JavaScript 3D physics engines
  - comparing / Comparing JavaScript 3D physics engines
  - concepts / Comprehending the physics engine concepts
- jigLib.JBox class / Using a rigid body (collider) for each scene object
- jigLib.JBox colliders / Picking using ray casting
- jigLib.JSegment class / Changing the color of the selected object
- jigLib.JSphere object / Adding gravity and a rigid body to the game scene
- jigLib.Plane rigid body / Adding gravity and a rigid body to the game scene
- JigLibJS
  - about / JigLibJS
  - constraints / JigLibJS
- join event / Understanding events and the code flow
- Joint DOFs
  - about / Understanding forward kinematics
- Joint offset
  - about / Understanding forward kinematics
- JRay class / Understanding the basic ray casting concepts
- JSegment class / Understanding the basic ray casting concepts
- JSON
  - 3DS files, converting to / Converting FBX/Collada/3DS files to JSON
  - FBX, converting to / Converting FBX/Collada/3DS files to JSON
  - Collada, converting to / Converting FBX/Collada/3DS files to JSON
- JSON faces array
  - parsing / Parsing the JSON faces array
- JSON file / The JSON file
  - with UV coordinates / Understanding the JSON file with UV coordinates
  - encoding / Understanding JSON file encoding
- JSON model
  - loading / Loading the JSON model
- JSON parser
  - changes / Changes in our JSON parser
- JTerrain object / Understanding the physics shapes

K

- keyboard interaction
  - adding / Adding keyboard and mouse interactions
- key combinations
  - Left arrow / Understanding the free camera
  - Right arrow / Understanding the free camera
  - S / Understanding the free camera
  - W / Understanding the free camera
  - Up arrow / Understanding the free camera
  - Down arrow / Understanding the free camera
  - Shift + A / Understanding the free camera
  - Shift + left arrow / Understanding the free camera
  - Shift + D / Understanding the free camera
  - Shift + right arrow / Understanding the free camera
  - Shift + up arrow / Understanding the free camera
  - Shift + down arrow / Understanding the free camera
- kinematics
  - about / Understanding forward kinematics

L

- Lambertian model
  - about / Understanding the illumination/reflection model
- Lambertian reflectance
  - about / Lambertian reflectance/diffuse reflection
- Lambertion reflection model
  - Gouraud shading on / Implementing Gouraud shading on a Lambertian reflection model
- lamps
  - used, for lighting up scenes / Lighting up the scene with lamps
  - vertex shader / The vertex shader, The vertex shader
  - fragment shader / The fragment shader, The fragment shader
  - adding / Adding multiple lamps
  - main code / Understanding the main code
- left-hand rotation
  - linear interpolation, using for / Using linear interpolation for left-hand rotation
- Left arrow / Understanding the free camera
- level parameter / Loading textures
- Light.js
  - implementing / Implementing Light.js
  - applying / Applying Lights.js
- lights
  - directional light / Directional lights
  - point light / Point lights
  - spotlight / Spotlights
  - Ambient lighting / Ambient lighting
  - Diffuse reflection / Diffuse reflection (Lambertian reflection)
  - Specular reflection / Specular reflection
  - rendering without / Rendering without light
  - multiple lights / Multiple lights and shaders
- lights object / Understanding the main code
- linear animation / Simple bullet action – linear animation
- linear interpolation / Linear interpolation, Linear interpolation
  - using, for grenade action / Using B-spline interpolation for grenade action
  - using, for left-hand rotation / Using linear interpolation for left-hand rotation

- linear transformations / Classifying into linear and affine transformations
- lineVel (linear velocity) / Implementing the free camera
- loadObject function / Implementing StageObject.js, Implementing the RiggedMesh class
- loadStageObject() function / Rendering our plane geometry
- loadStageObject function / Using linear interpolation for left-hand rotation, Rendering our plane geometry
- loadTexture() function / Rendering our plane geometry
- loggedIn event / The server code
- lookAt function / Using the lookAt function
  - using / Using the lookAt function
  - about / Implementing the basic camera
- look at position setting / Understanding the free camera

M

- magnification filtering / Applying filtering modes
- material file format (MTL
  - about / Understanding the material file format (MTL)
- materials
  - about / Supporting multiple objects
- matrices
  - about / Matrices
- matrixAutoUpdate / Enhancing the StageObject class
- matrixWorld / Enhancing the StageObject class
- matrixWorldNeedsUpdate / Enhancing the StageObject class
- MD5Anim
  - about / Loading MD5Mesh and MD5Anim files
- MD5Mesh
  - about / Loading MD5Mesh and MD5Anim files
- measureText function / Using the sprite texture
- mesh
  - about / Understanding mesh, polygon, and vertices
- message / The client socket's reserved events
- messageObj.username parameter / The client code
- message property / The server code
- mimeTypes array / The server code
- minification filtering / Applying filtering modes
- mipmapping
  - about / Understanding mipmapping
  - implemeting / Implementing mipmapping
  - nearest-neighbor interpolation / Nearest-neighbor interpolation
  - linear interpolation / Linear interpolation
  - nearest-neighbor with mipmapping / Nearest-neighbor with mipmapping
  - bilinear filtering / Bilinear filtering with mipmapping
  - Trilinear filtering / Trilinear filtering
- MIRRORED_REPEAT, wrapping mode / Texture wrapping
- model labels
  - 2D textures, adding as / Adding 2D textures as model labels
- modelMatrix / Enhancing the StageObject class
- models
  - exporting, from Blender / Exporting models from Blender

    - exporting, from 3D software in JSON / Exporting models from 3D software in JSON
- ModelSprite class
    - about / Implementing the ModelSprite class
    - main flow code / Understanding the main flow code
- model transformation
    - applying / Applying the model transformation
- ModelView transformation / ModelView transformation
- ModelView transformations
    - about / Understanding ModelView transformations
    - model transformation, applying / Applying the model transformation
    - view transformation / Understanding the view transformation
    - lookAt function, using / Using the lookAt function
    - camera rotation / Understanding the camera rotation
    - quaternions, using / Using quaternions
- modifyGeometry function / Extending our terrain with physics
- morphedVertexNormalsFromObj() function / Understanding a simple terrain – plane geometry
- morphedVertexNormalsFromObj function / Implementing Geometry.js
- mouse events
    - handling / Handling mouse events
- mouse interaction
    - adding / Adding keyboard and mouse interactions
- moveForward function / Implementing the free camera
- mtllib, Box.obj file element / Exporting models from Blender
- multi-texturing
    - about / Multi-texturing
- multiplayer game
    - implementing / Implementing a multiplayer game
    - events / Understanding events and the code flow
    - server code / The server code
    - client code / The client code
- multiple objects
    - changes, summarizing / Implementing Stage.js
- mvMatrix / Understanding ModelView transformations
- mvMatrix states
    - using / Using mvMatrix states

N

- nearest-neighbor interpolation / Nearest-neighbor interpolation
- nearest-neighbor with mipmapping / Nearest-neighbor with mipmapping
- Node.js
  - using, for multiplayer games / Using Node.js and Socket.IO for multiplayer games
  - URL, for installing / Using Node.js and Socket.IO for multiplayer games
  - used, for implementing HTTP server / Implementing the HTTP server using Node.js
- node modules
  - URL / Using Node.js and Socket.IO for multiplayer games
- normal / Understanding surface normals for lighting calculations
- normalize() function / Implementing Gouraud shading – Blinn-Phong reflection
- Normalized Device Coordinates (NDC) / Calculating the screen coordinates of a click
- normals
  - about / Supporting multiple objects
- normal transformation
  - about / Understanding normal transformation
- norm parameter / Associating buffer objects with shader attributes

O

- o, Box.obj file element / Exporting models from Blender
- object materials
  - about / Understanding object materials
- objects
  - exported from Blender, rendering / Rendering objects exported from Blender
- object space / ModelView transformation
- OBJ file
  - converting, to JSON file format / Converting the OBJ file to the JSON file format
- offscreen rendering
  - framebuffers used / Offscreen rendering using framebuffers
  - texture object, creating / Creating a texture object to store color information
  - renderbuffer, creating / Creating a renderbuffer for depth information
  - texture, associating to framebuffers / Associating a texture and a renderbuffer to framebuffers
  - renderbuffer, associating to framebuffers / Associating a texture and a renderbuffer to framebuffers
- offsetLeft variable / Calculating the screen coordinates of a click
- offset parameter / Associating buffer objects with shader attributes
- offsetTop variable / Calculating the screen coordinates of a click
- onMouseDown event / Handling mouse events
- onmousedown event / Changing the color of the selected object
- onMouseMove event / Handling mouse events
- orbit camera
  - about / Comprehending the orbit camera
  - parameters / Comprehending the orbit camera
  - implementing / Implementing the orbit camera
  - closestDistance parameter / Implementing the orbit camera
  - FarthestDistance parameter / Implementing the orbit camera
  - orbitPoint parameter / Implementing the orbit camera
  - pitch function / Understanding the pitch function for the orbit camera
  - yaw function / Understanding the yaw function for the orbit camera
  - using / Using an orbit camera
- orbitPoint parameter / Implementing the orbit camera
- out.frac variable / Understanding the basic ray casting concepts

- out.position variable / Understanding the basic ray casting concepts
- out.rigidBody variable / Understanding the basic ray casting concepts
- out parameter / Understanding the basic ray casting concepts

P

- param parameter / Loading textures
- parent object / Enhancing the StageObject class
- parse function / Loading the animation data
- parseJSON.js
  - implementing / Implementing parseJSON.js
- parseJSON function / Implementing Stage.js
- parseJSON object / Implementing StageObject.js
- per-vertex operation / Vertex transformation with a basic vertex shader
- perspective divide / Unproject the vector
- perspective transformations
  - about / Understanding perspective transformations
  - viewing frustum / Understanding the viewing frustum
  - viewing frustum, defining / Defining the view frustum
- per vertex colors
  - about / Supporting multiple objects
- Phong reflectance model
  - about / Understanding the illumination/reflection model
- Phong shading
  - about / Phong shading
  - implementing / Implementing Phong shading – Blinn-Phong reflection
- phpwebsocket
  - about / Understanding the WebSockets server
- physics
  - objects / Learning about objects in the physics system
  - terrain, extending with / Extending our terrain with physics
- physics, objeects
  - particles / Particles
  - rigid body / Rigid bodies
  - soft body / Soft bodies
- physics shapes
  - about / Understanding the physics shapes
- picking
  - about / Learning the basics of picking
  - objects color based / Picking based on an object's color
  - ray casting used / Picking using ray casting
  - implementing, ray casting used / Implementing picking using ray casting

- picking, ray casting used
  - rigid body (collider), using for scene object / Using a rigid body (collider) for each scene object
  - click, screen coordinates / Calculating the screen coordinates of a click
  - vector, unprojecting / Unproject the vector
  - ray segment, creating / Creating a ray segment
  - intersection, checking / Checking for an intersection
  - selected object, color changing / Changing the color of the selected object
- pitch function, orbit camera / Understanding the pitch function for the orbit camera
- pixels / The fragment shader
- pixels parameter / Loading textures
- PixPlant
  - about / Texturing basics
- plane geometry
  - about / Understanding a simple terrain – plane geometry
  - equation / Understanding a simple terrain – plane geometry
  - diagram / Understanding a simple terrain – plane geometry
  - rendering / Rendering our plane geometry
- pname parameter / Loading textures
- point light
  - about / Point lights
- polling / Communicating in real time
- polygon
  - about / Understanding mesh, polygon, and vertices
- polynomial interpolation / Polynomial interpolation
- positional lights
  - about / Understanding positional lights
- position setting / Understanding the free camera
- powerOfTwo function / Using the sprite texture
- primitive
  - drawing / Drawing our primitive
- primitive assembly stage / The fragment shader
- projection transformation / Projection transformation

Q

- quaternions
  - using / Using quaternions

R

- Radius parameter / Comprehending the orbit camera
- rasterization / The fragment shader
- RayCaster class / Checking for an intersection, Changing the color of the selected object
- ray casting
  - about / Understanding the basic ray casting concepts
  - used, for picking / Picking using ray casting
- ray segment
  - creating / Creating a ray segment
- reconnect / The client socket's reserved events
- reconnecting / The client socket's reserved events
- reconnect_failed / The client socket's reserved events
- redrawWithClampingMode function / The redrawWithClampingMode function
- removeFromUpdate function / Loading the animation data
- renderbuffer
  - creating, to depth information / Creating a renderbuffer for depth information
  - associating, to framebuffers / Associating a texture and a renderbuffer to framebuffers
- rendering pipeline, WebGL
  - about / Understanding WebGL's rendering pipeline
  - Framebuffers / Framebuffers
- REPEAT, wrapping mode / Texture wrapping
- request animation frames
  - about / Understanding request animation frames
- requestAnimFrame function / Understanding request animation frames
- RGB (Red, Green, and Blue) space / Learning more about colors
- rigged JSON model
  - loading / Loading a rigged JSON model, Loading the rigged model
  - JSON file encoding / Understanding JSON file encoding
  - animating / Animating a rigged JSON model
  - animation data / JSON model – animation data
  - animation data, loading / Loading the animation data
- RiggedMesh class
  - implementing / Implementing the RiggedMesh class
  - this.bones[] array / Implementing the RiggedMesh class

    - this.boneMatrices[] array / Implementing the RiggedMesh class
    - this.skinIndexBuffer variable / Implementing the RiggedMesh class
    - this.skinWeightBuffer variable / Implementing the RiggedMesh class
- RiggedMesh object / Loading the animation data
- rigged model
    - StageObject class, enhancing / Enhancing the StageObject class
    - bone class, implementing / Implementing the bone class
    - RiggedMesh class, implementing / Implementing the RiggedMesh class
    - skinned model, loading / Loading the skinned model
    - loading / Loading the skinned model
- Right arrow / Understanding the free camera
- rigid body
    - about / Rigid bodies
- RigidBody class / Diving deep into collision detection
- robotic bone hierarchy
    - about / Comprehending the joint hierarchy
- roomJoined event / Understanding events and the code flow
- rotateOnAxis function / Implementing the free camera

S

- S / Understanding the free camera
- s, Box.obj file element / Exporting models from Blender
- scene
  - loading / Loading the scene
  - lighting up, with lamps / Lighting up the scene with lamps
  - textures, applying / Applying textures to our scene, Applying a texture to the scene
- sendRoomMessage function / The client code
- serverData / Understanding Socket.IO
- serverMessage event / Understanding events and the code flow
- setDistance function / Understanding the yaw function for the orbit camera
- setInterval function / Understanding request animation frames
- setLightUniform() function / Understanding the main code
- setLightUniform function / Understanding the main code
- setLookAtPoint function / Implementing the free camera
- setMatrixUniforms / Understanding ModelView transformations
- setMatrixUniforms() function / Drawing using vertex buffer objects
- setPosition function / Implementing the orbit camera
- setTimeout function / Understanding request animation frames
- shader attributes
  - buffer objects, associating with / Associating buffer objects with shader attributes
- shader code, cubemaps / Understanding the shader code
- shader code, cube maps / Understanding the shader code
- shaders
  - about / Shaders
  - vertex shader / The vertex shader
  - fragment shader / The fragment shader
  - multiple shaders / Multiple lights and shaders
  - loading / Loading and linking shaders
  - linking / Loading and linking shaders
- shader variables
  - about / Shader variable qualifiers
  - attributes qualifier / Attributes
  - uniforms qualifier / Uniforms
  - uniforms / Uniforms
  - varying qualifier / The varying qualifier

    - compiling / Compiling and linking shaders
    - linking / Compiling and linking shaders
- shading models
    - flat shading / Flat shading
    - Gouraud shading / Gouraud shading
    - Phong shading / Phong shading
- Shift + A / Understanding the free camera
- Shift + D / Understanding the free camera
- Shift + down arrow / Understanding the free camera
- Shift + left arrow / Understanding the free camera
- Shift + right arrow / Understanding the free camera
- Shift + up arrow / Understanding the free camera
- simple skinning
    - about / Simple skinning
- simulation loop
    - updating / Updating the simulation loop
- size parameter / Associating buffer objects with shader attributes
- skeletons
    - about / Understanding the basics of a character's skeleton
- SkinIndex parameter / Loading the skinned model
- skinMatrix variable / Implementing the bone class
- skinned animation
    - about / A briefing on skinned animation
- skinned model
    - loading / Loading the skinned model
- skinning
    - simple skinning / Simple skinning
    - smooth skinning / Smooth skinning
- skinWeight parameter / Loading the skinned model
- smooth skinning
    - about / Smooth skinning
    - binding matrix / The binding matrix
    - final vertex transformation / The final vertex transformation
    - final normal transformation / The final normal transformation
- Sobel operator
    - URL / Applying filters using framebuffers
- socket / Learning the Socket.IO API
- Socket.IO
    - about / Understanding the WebSockets server, Understanding

Socket.IO
    - using, for multiplayer games / Using Node.js and Socket.IO for multiplayer games
    - io.sockets / Learning the Socket.IO API
    - socket / Learning the Socket.IO API
    - client socket, reserved events / The client socket's reserved events
    - user data, storing on server side / Storing user data on the server side
- Socket.IO, rooms
    - joining / Leaving and joining
    - leaving / Leaving and joining
    - event, broadcasting / Broadcasting an event in the room
    - information, getting / Getting the room's information
- socket object / Broadcasting an event in the room
- soft body
    - about / Soft bodies
- specular component / Understanding the illumination/reflection model
- Specular reflection / Specular reflection
- spline interpolation / Spline interpolation
- spotlight
    - about / Spotlights
- Sprite class
    - implementing / Implementing the Sprite class
    - about / Implementing the Sprite class
- sprite texture
    - using / Using the sprite texture
- SpriteTexture class / Implementing the Sprite class
- square
    - coloring / Coloring our square
- square geometry
    - using / Using a square geometry
- SquareGeometry class / Understanding the square geometry code
- square geometry code
    - about / Understanding the square geometry code
- Stage.js
    - implementing / Implementing Stage.js
- stage class / Implementing Stage.js
- stageObject / Enhancing the StageObject class
- StageObject.js
    - implementing / Implementing StageObject.js

- StageObject class / Implementing StageObject.js, Implementing Stage.js, Adding gravity and a rigid body to the game scene, Using a rigid body (collider) for each scene object, Implementing the Sprite class
  - enhancing / Enhancing the StageObject class
- start() function / Drawing using vertex buffer objects, Adding gravity and a rigid body to the game scene, Changing the color of the selected object
- start function / Understanding the main code
- state machine
  - about / Understanding WebGL – a state machine
- STATIC_DRAW / Vertex buffer objects – uploading data to GPU
- stream
  - URL / Using Node.js and Socket.IO for multiplayer games
- STREAM_DRAW / Vertex buffer objects – uploading data to GPU
- stride parameter / Associating buffer objects with shader attributes
- superc function / Implementing the free camera
- surface normal
  - about / Understanding surface normals for lighting calculations
  - calculating, from vertices / Calculating normals from vertices and indices
  - calculating, from indices / Calculating normals from vertices and indices
- system.integrate() function / Adding gravity and a rigid body to the game scene

T

- target parameter / Loading textures
- terrain
  - about / Understanding a simple terrain – plane geometry
  - extending, with physics / Extending our terrain with physics
- TerrainData object / Extending our terrain with physics
- texImage2D function / The fragment shader
- texture
  - associating, to framebuffers / Associating a texture and a renderbuffer to framebuffers
- texture animation
  - using, for explosion effect / Using texture animation for an explosion effect
- texture buffer / Applying filtering modes
- textured object
  - loading / Loading a textured object
- texture filtering
  - about / Comprehending texture filtering
- textureIndex property / Working with the control code
- texture magnification
  - about / Comprehending texture filtering
- texture minification / Comprehending texture filtering
- texture parameter / Loading textures
- textures
  - 2D textures / Texturing basics
  - 3D textures / Texturing basics
  - loading / Loading textures, The fragment shader
  - sampler data type / A new data type – sampler
  - lookup functions / A new data type – sampler
  - applying, to square / Applying a texture to the square
  - applying, to scene / Applying textures to our scene, Applying a texture to the scene
  - canvas 2D, using / Using canvas 2D for textures
- texture wrapping
  - about / Texture wrapping
  - HTML / The HTML
  - event handlers / The event handlers
  - redrawWithClampingMode function / The redrawWithClampingMode

function
- this.boneMatrices[] array / Implementing the RiggedMesh class
- this.bones[] array / Implementing the RiggedMesh class
- this.initializePhysics() function / Revisiting the grenade and bullet actions
- this.pose() function / Implementing the RiggedMesh class
- this.power variable / Using the sprite texture
- this.skinIndexBuffer variable / Implementing the RiggedMesh class
- this.skinWeightBuffer variable / Implementing the RiggedMesh class
- tick function / Understanding request animation frames, Understanding the main code
- time-based animation / Understanding time-based animation, Implementing time-based animation
- Torrance-Sparrow model
  - about / Understanding the illumination/reflection model
- totalElapsedTime variable / Understanding the main code
- transform() function / Understanding canvas 2D basics and the drawing API
- transformations
  - about / Understanding transformations
  - linear transformations / Classifying into linear and affine transformations
  - affine transformations / Classifying into linear and affine transformations
  - to render 3D objects / Understanding transformations required to render 3D objects
  - ModelView transformation / ModelView transformation
  - projection transformation / Projection transformation
- translate function / Handling mouse events
- translateOnAxis function / Enhancing the StageObject class
- traverse function / Enhancing the StageObject class
- TRIANGLE_XXX option / Drawing our primitive
- Trilinear filtering / Trilinear filtering
- type parameter / Associating buffer objects with shader attributes, Loading textures

U

- uniformMatrix3fv function / Implementing Gouraud shading on a Lambertian reflection model
- uniform qualifier
  - about / Uniforms
- unloopedCurrentTime value / Loading the animation data
- Up arrow / Understanding the free camera
- update function / Implementing the bone class
- updateMatrix function / Using a rigid body (collider) for each scene object
- updateMatrixWorld function / Enhancing the StageObject class
- useSkinning parameter / Loading the skinned model
- UV coordinates
  - parsing, from JSON file / Parsing UV coordinates from the JSON file
  - vertices / Revisiting vertices, normals, and the indices array
  - normals / Revisiting vertices, normals, and the indices array
  - indices array / Revisiting vertices, normals, and the indices array
  - texture coordinates, restructuring for / Restructuring for texture coordinates
  - algorithms, to create new arrays / Algorithm one to create new arrays, Algorithm two to create new arrays
  - algorithm, dry run used / Understanding the algorithm using a dry run
- UV map
  - about / Supporting multiple objects

V

- v, Box.obj file element / Exporting models from Blender
- varyings qualifier
  - about / The varying qualifier
- vector
  - unprojecting / Unproject the vector
- vectors
  - about / Vectors
- vertex
  - about / Understanding mesh, polygon, and vertices
- vertex buffer objects
  - about / Vertex buffer objects – uploading data to GPU
  - used, for drawing / Drawing using vertex buffer objects
- vertex color
  - used, for coloring / Coloring using the vertex color
- vertexNormals array / Implementing Face.js
- vertex shader
  - about / The vertex shader, The vertex shader, The vertex shader, The vertex shader
  - used, for vertex transformation / Vertex transformation with a basic vertex shader / The vertex shader
- vertex shader code
  - implementing / Implementing the vertex shader code
- vertices
  - about / Understanding mesh, polygon, and vertices, Supporting multiple objects
- verticesFromFaceUvs() function / Understanding a simple terrain – plane geometry
- verticesFromFaceUvs function / Algorithm one to create new arrays
- viewing frustum
  - about / Understanding the viewing frustum, Defining the view frustum
- view matrix
  - about / Understanding the view transformation
  - breakdown / Comprehending the components of a camera matrix
  - and camera transformation matrix, conversions / Converting between the camera matrix and view matrix
- view transformation

    - about / Understanding the view transformation
- vn, Box.obj file element / Exporting models from Blender
- vt, Box.obj file element / Exporting models from Blender
- vTextureCoord variable / The fragment shader

W

- W / Understanding the free camera
- Ward's anisotropic model
  - about / Understanding the illumination/reflection model
- Wavefront (OBJ) format
  - about / Understanding and loading the Wavefront (OBJ) format
- Wavefront object file / Rendering 3D objects
- WebGL
  - about / Understanding WebGL
  - differentiating, from game engine / Differentiating WebGL from the game engine
  - rendering pipeline / Understanding WebGL's rendering pipeline
- WebGL API
  - about / A walkthrough of the WebGL API
- WebGL application
  - debugging / Debugging a WebGL application
- WebGL context
  - initializing / Initializing the WebGL context
- WebGL texture mapping
  - principles / The challenge and the algorithm
- WebSocket API
  - about / Understanding the WebSocket API
- WebSockets
  - about / Understanding WebSockets
  - WebSocket API / Understanding the WebSocket API
  - server / Understanding the WebSockets server
- WebSockets server
  - about / Understanding the WebSockets server
  - phpwebsocket / Understanding the WebSockets server
  - Socket.IO / Understanding the WebSockets server
- window.requestAnimationFrame() function / Understanding request animation frames
- worldMatrix / Enhancing the StageObject class
- wss protocol / Understanding the WebSocket API

Y

- yaw function, orbit camera / Understanding the yaw function for the orbit camera

Z

- z-fighting / Implementing the basic camera